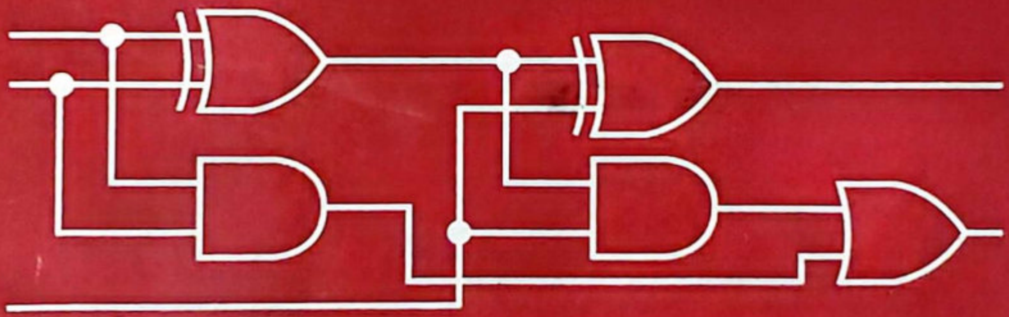
Digital Logic Design



Md. Mozammel Huq Azad Khan

University Grants
Commission of Bangladesh

Digital Logic Design has been written as a textbook for bachelors programs of computer engineering, computer science, electrical engineering, communications engineering, electronics, information technology, and other disciplines where digital logic design is the part of the curriculum. In the book, basic techniques and procedures of logical level design of digital systems are discussed. The techniques are presented from the classical point of view. Boolean algebra and truth tables are used for the analysis and design of combinational circuits and state transition techniques for the analysis and design of sequential circuits. Register transfer logic design and digital systems modeling and simulation using Verilog hardware description language are also emphasized. The book has been systematically organized and the presentation has been kept at a level suitable for a student with or without a basic knowledge of digital electronics. Every chapter includes a set of exercises. At the end of each chapter, a list of further readings is added. The last chapter of the book outlines experiments that can be performed in the laboratory with hardware that is readily and inexpensively available commercially. The subject of the book is also known by other names such as logic design, digital logic, digital design, and switching circuits.

UGC Publication No. 93
ISBN 984-809-024-X

DIGITAL LOGIC DESIGN

PIAOPS

Md. Mozammel Huq Azad Khan

Professor of Computer Science and Engineering
East West University, Dhaka, Bangladesh
Former Professor of Computer Science and Engineering
Khulna University, Khulna, Bangladesh

University Grants Commission of Bangladesh
Agargaon, Dhaka-1207

First Published: April 2006

Published by : The University Grants Commination of Bangladesh
Agargaon, Dhaka, Bangladesh
Phone : 8122755, 9115197
E-mail : chairmanugc@yahoo.com
Website : www.ugc.gov.bd
UGC Publication No 107
ISBN 984-809-038-X

Copyright : ©reserved by the University Grants Commission of Bangladesh. No part of this publication may be reproduced or transmitted in any form or by any means without the prior written permission of the UGC.

Supervised by : K. M. Abdul Awwal
Additional Director &
Fakhrul Islam
Asst. Director, UGC.

Cover design by : Dr. Md. Mozammel Huq Azad Khan

Printed by : Mother Printers
8, 10 Nilkhet, Babupura, Dhaka
Tel : 8615959, 8622802

Price : Tk. 160.00
US$ 10.00
Euro 15.00

ISBN 984-809-024-X

Foreword

Today's society has been greatly impacted by the information revolution, which would have not been possible without the use of digital techniques in the backend of the information technology. Computers and communication technologies are the two that have made possible unbelievable explosion of the current information technology. Both of these technologies are extensively using digital electronic components. In fact in every aspect of machine based activities, starting from home appliances and entertainment equipment to computers, communication devices and industrial automation, digital electronics has been ubiquitously used.

Digital technology requires three tires of design expertise. At the low level is the digital electronics, which deals with the device level implementations of the digital circuits. The mid-level design deals with the logical level design of the digital circuits and commonly known as digital logic design. The top-level design deals with architectural-level design of a complex digital system. This book deals with the theories and practices of logical level design of a digital circuit. This level of expertise is essential for students of computer science, computer engineering, electrical engineering, electronics, communications engineering, information technology, industrial automation, and other disciplines where digital technology is a part of the curriculum. This book has been written as a textbook for the undergraduate students of these disciplines. Professor Dr. Md. Mozammel Huq Azad Khan, Dean, Faculty of Sciences and Engineering, East West University, Bangladesh has contributed his sincere efforts towards writing this book based on his more than 20 years of experience of classroom teaching the subject at undergraduate and graduate levels in various public and private universities.

The University Grants Commission of Bangladesh is delighted to publish such a valuable textbook. We expect that the book will be of immense benefit to the students and teachers of the engineering disciplines requiring knowledge of digital logic design. Sincere thanks are due to the officers and staffs of the Publication Section of the Commission for undertaking pains to publish the book.

Professor Dr. M. Asaduzzaman
Chairman
University Grants Commission of Bangladesh

CONTENTS

PREFACE XVII
ORGANIZATION OF THE BOOK XIX
LESSON PLAN XXI

1 INTRODUCTION TO DIGITAL SYSTEMS 1
1.1 Introduction 1
1.2 Digital and Analog Systems 1
1.3 Binary Numbers in Digital Systems 2
1.4 Binary Logic 3
  1.4.1 Definition of Binary Logic 3
  1.4.2 Truth Table 4
  1.4.3 Switching Circuits 4
  1.4.4 Logic Gates 5
1.5. Integrated Circuits 6
Exercises 6
Further Readings 6

2 NUMBER SYSTEMS AND CODES 7
2.1 Introduction 7
2.2 Number Systems 7
2.3 Complements 8
  2.3.1 The *r*'s Complement 8
  2.3.2 The (*r* – 1)'s Complement 9
  2.3.3 Subtraction with *r*'s Complement 9
  2.3.4 Subtraction with (*r* – 1)'s Complement 10
2.4 Binary Number System 12
  2.4.1 Binary Number Representation 12
  2.4.2 Binary to Decimal Conversion 12
  2.4.3 Decimal to Binary Conversion 13
  2.4.4 2's Complement of Binary Numbers 14
  2.4.5 1's Complement of Binary Number 14
2.5 Binary Arithmetic 14
  2.5.1 Binary Addition 15
  2.5.2 Binary Subtraction 15
  2.5.3 Binary Multiplication 16
  2.5.4 Binary Division 16
  2.5.5 Signed Binary Numbers 16
  2.5.6 1's Complement Binary Arithmetic 19
  2.5.6 2's Complement Binary Arithmetic 21
2.6 Octal Number System 22
  2.6.1 Octal Number Representation 22
  2.6.2 Octal to Decimal Conversion 22
  2.6.3 Decimal to Octal Conversion 23

2.6.4 Octal to Binary Conversion 23
2.6.5 Binary to Octal Conversion 24
2.6.6 Decimal to Binary Conversion via Octal and Vice Versa 24
2.6.7 Octal Arithmetic 25
2.6.8 Application of Octal Number System 26
2.7 Hexadecimal Number System 26
2.7.1 Hexadecimal Number Representation 26
2.7.2 Hexadecimal to Decimal Conversion 26
2.7.3 Decimal to Hexadecimal Conversion 27
2.7.4 Hexadecimal to Binary Conversion 27
2.7.5 Binary to Hexadecimal Conversion 28
2.7.6 Hexadecimal to Octal Conversion and Vice Versa 28
2.7.7 Decimal to Binary Conversion via Hexadecimal and Vice Versa 28
2.7.8 Hexadecimal Arithmetic 29
2.7.9 Application of Hexadecimal Number System 30
2.8 Binary Codes 30
2.8.1 BCD Code 30
2.8.2 Excess-3 Code 31
2.8.3 Gray Code 31
2.8.4 Error-Detection Code 35
2.8.5 ASCII Character Code 39
Exercises 40
Further Readings 43

3 BOOLEAN ALGEBRA 44
3.1 Introduction 44
3.2 Axiomatic Definition of Boolean Algebra 44
3.2.1 Postulates of Boolean Algebra 44
3.2.2 Two-valued Boolean Algebra 45
3.3 Basic Theorems and Properties of Boolean Algebra 47
3.3.1 Duality Principle 47
3.3.2 Basic Theorems of Boolean Algebra 47
3.3.3 Operator Precedence 51
3.4 Boolean Function 51
3.4.1 Algebraic Expression of Boolean Function 51
3.4.2 Truth Table Representation of Boolean Function 51
3.4.3 Non-unique Algebraic Expressions of Boolean Function 52
3.4.4 Logic Diagram 53
3.4.5 Complement of Boolean Function 54
3.5 Forms of Boolean Functions 54
3.5.1 Minterms and Maxterms 54
3.5.2 Canonical Forms 55
3.5.3 Noncanonical Forms 58
3.5.4 Nonstandard Forms 59
3.6 Incompletely Specified Boolean Function 60
Exercises 61
Further Readings 62

4 LOGIC GATES 64
4.1 Introduction 64
4.2 Logic Operations 64
4.3 Logic Gates 66
4.4 Universality of NAND and NOR Gates 72
4.5 Alternative Graphic Symbols for NAND, NOR, and NOT Gates 73
4.6 EXOR and EXNOR Gates 74
4.6.1 Properties of EXOR and EXNOR Operations 74
4.6.2 EXOR and EXNOR Gate Implementations 75
4.7 Integrated Circuit Logic Gates 76
4.7.1 IC Logic Families 76
4.7.2 Logic Conventions 78
4.7.3 Characteristics of IC Logic Families 79
4.8 Some Available TTL Logic Gates 80
Exercises 82
Further Readings 83

5 SIMPLIFICATION OF BOOLEAN FUNCTIONS 84
5.1 Introduction 84
5.2 Algebraic Simplification of Boolean Functions 85
5.3 Prime Implicants 86
5.4 Karnaugh Map Method for Simplification of Boolean Functions 87
5.4.1 The Karnaugh Map 87
5.4.2 Representation of Truth Table on a K-map 89
5.4.3 Representation of Sum of Products Function on a K-map 90
5.4.4 Representation of Product of Sums Function on a K-map 91
5.4.5 Grouping of Adjacent 1s on a K-map 92
5.4.6 Grouping of Adjacent 0s on a K-map 96
5.4.7 Simplification of Sum of Products Function using K-map 97
5.4.8 Simplification of Products of Sums Function using K-map 102
5.4.9 Simplification of Sum of Products vs. Product of Sums Function 104
5.5 Quine-McCluskey Method for Simplification of Boolean Function 104
5.5.1 The Quine-McCluskey method 104
5.5.2 Determination of All Prime Implicants 105
5.5.3 Selection of Prime Implicants 110
5.5.4 Cyclic Prime Implicant Chart 113
5.6 Iterated Consensus Method for Simplification of Boolean Function 114
5.6.1 The Consensus Operation 114
5.6.2 Simplification of Boolean Function 115
Exercises 118
Further Readings 119

6 COMBINATIONAL LOGIC DESIGN 120
6.1 Introduction 120
6.2 Design of Combinational Circuits 120
6.3 Analysis of Combinational Circuits 123
6.4 Two-Level NAND Circuits 126
6.5 Two-Level NOR Circuits 128

6.6 Multi-Level NAND Circuits 130
6.6.1 Boolean Function Implementation with Multi-Level NAND Circuits 130
6.6.2 Analysis of Multi-Level NAND Circuits 132
6.7 Multi-Level NOR Circuits 136
6.7.1 Boolean Function Implementation with Multi-Level NOR Circuits 136
6.6.2 Analysis of Multi-Level NOR Circuits 137
6.8 Two-Level AND-EXOR Logic Circuits 140
6.9 Three-Level AND-OR-EXOR Logic Circuits 144
Exercises 147
Further Readings 150

7 ARITHMETIC AND COMPARATOR CIRCUITS 151
7.1 Introduction 151
7.2 Adders 152
7.2.1 Half-Adder 152
7.2.2 Full-Adder 153
7.2.3 Full-Adder using Half-Adder 154
7.3 Subtractor 155
7.3.1 Half-Subtractor 156
7.3.2 Full-Subtractor 157
7.4 Binary Parallel Adder 159
7.5 Binary Adders with Look-Ahead Carry 161
7.6 2's Complement Arithmetic Circuits 164
7.6.1 2's Complement Adder 164
7.6.2 2's Complement Subtractor 165
7.6.3 Combined 2's Complement Adder/Subtractor 166
7.7 BCD Arithmetic Circuits 166
7.7.1 BCD Adder 167
7.7.2 BCD Subtractor 170
7.7.3 Combined BCD Adder/Subtractor 170
7.8 Available MSI Arithmetic ICs 171
7.8.1 Look-Ahead Carry Generator 171
7.8.2 4-bit Binary Adder 173
7.9 Magnitude Comparator 173
Exercises 176
Further Readings 176

8 ENCODERS, DECODERS, MULTIPLEXERS, AND DEMULTIPLEXERS 177
8.1 Introduction 177
8.2 Encoders 177
8.2.1 Design of Encoders 177
8.2.2 Available MSI Encoder ICs 179
8.3 Decoders 180
8.3.1 Design of Decoders 180
8.3.2 Decoders with Enable Input 184
8.3.3 Expansion of Decoder 185

8.3.4 Combinational Logic Implementation with Decoder 187
8.3.5 Available MSI Decoder ICs 189
8.4 Code Converters 189
8.5 BCD-to-7-Segment Display Decoder 192
8.5.1 Design of BCD-to-7-Segment Display Decoder 192
8.5.2 Available MSI BCD-to-7-Segment Display Decoder/Driver ICs 195
8.6 Parity Generator and Checker 195
8.6.1 Design of Parity Generator and Checker 195
8.6.2 Available MSI Parity generator/Checker IC 198
8.7 Multiplexers 198
8.7.1 Design of Multiplexers 198
8.7.2 Multiplexer with Enable Input 200
8.7.3 Available MSI Multiplexer ICs 201
8.7.4 Expansion of Multiplexer 201
8.7.5 Combinational Logic Implementation with Multiplexers 203
8.8 Demultiplexers 206
8.8.1 Design of Demultiplexers 206
8.8.2 Available MSI Demultiplexer ICs 208
8.8.3 Expansion of Demultiplexer 208
8.8.4 Combinational Logic Implementation with Demultiplexer 209
Exercises 210
Further Readings 212

9 FLIP-FLOPS 213
9.1 Introduction 213
9.2 A 1-Bit Memory Cell 214
9.3 S-R Flip-Flop 216
9.3.1 Basic S-R Flip-Flop 216
9.3.2 S-R Flip-Flop with Preset and Clear Inputs 219
9.4 J-K Flip-Flop 220
9.4.1 Basic J-K Flip-Flop 220
9.4.2 The Race-Around Condition in J-K Flip-Flop 222
9.4.3 The Master-Slave J-K Flip-Flop 223
9.5 D Flip-Flop 224
9.6 T Flip-Flop 226
9.7 Conversion From One Type of Flip-Flop to Another Type 227
9.8 Triggering of Flip-Flops 228
9.9 Available Flip-Flop ICs 230
9.10 Applications of Flip-Flops 230
9.10.1 Bounce-Elimination Switch 230
9.10.2 Random-Access Memory 232
9.10.3 Frequency Divider 233
Exercises 234
Further Readings 235

10 SEQUENTIAL LOGIC DESIGN 236
10.1 Introduction 236
10.2 Representation of Sequential Circuits 236

10.3 Analysis Procedure of Sequential Circuits 240
10.4 State Reduction and Assignment 242
  10.4.1 State Reduction 242
  10.4.2 State Assignment 244
10.5 Design Procedure of Sequential Circuit 246
Exercises 257
Further Readings 260

11 REGISTERS AND COUNTERS 261
11.1 Introduction 261
11.2 Registers and Their Classification 261
11.3 Serial-In Serial-Out Registers 262
  11.3.1 Serial-In Serial-Out Right-Shift Register 262
  11.3.2 Serial-In Serial-Out Left-Shift Register 264
  11.3.3 Serial-In Serial-Out Bi-Directional Shift Register 264
11.4 Serial-In Parallel-Out Registers 265
  11.4.1 Serial-In Parallel-Out Right-Shift Register 265
  11.4.2 Serial-In Parallel-Out Left-Shift Register 266
  11.4.3 Serial-In Parallel-Out Bi-Directional Shift Register 266
11.5 Parallel-In Parallel-Out Registers 266
11.6 Parallel-In Serial-Out Registers 268
  11.6.1 Parallel-In Serial-Out Right-Shift Register 268
  11.6.2 Parallel-In Serial-Out Left-Shift Register 270
  11.6.3 Parallel-In Serial-Out Bi-Directional Shift Register 271
11.7 Universal Register 271
11.8 Available MSI Register ICs 273
11.9 Application of Registers 274
11.10 Counters and Their Classification 274
11.11 Binary Asynchronous Counters 275
  11.11.1 Binary Asynchronous Up Counter 275
  11.11.2 Binary Asynchronous Down Counter 277
  11.11.3 Binary Asynchronous Up/Down Counter 278
  11.11.4 Binary Asynchronous Presettable Counter 279
  11.11.5 Binary Asynchronous Counter with Modulus Less Than 2^n 279
  11.11.6 Cascading of Binary Asynchronous Counters 284
11.12 Binary Synchronous Counters 285
  11.12.1 Binary Synchronous Up Counter 285
  11.12.2 Binary Synchronous Down Counter 286
  11.12.3 Binary Synchronous Up/Down Counter 287
  11.12.4 Binary Synchronous Presettable Counter 288
  11.12.5 Binary Synchronous Counter with Modulus Less Than 2^n 289
11.13 Odd Sequence Counter 291
11.14 Special Counters 293
  11.14.1 Ring Counter 293
  11.14.2 Twisted-Ring Counter 294

11.15 Available MSI Counter ICs 295
11.16 Application of Counters 296
  11.16.1 Word-Time Control 296
  11.16.2 Timing Signals Generator 298
  11.16.3 Pulse-Train Generator 299
  11.16.4 Digital Clock 300
Exercises 300
Further Readings 303

12 MEMORIES AND PROGRAMABLE LOGIC DEVICES 304
12.1 Introduction 304
12.2 Memories and Their Classifications 304
12.3 Random-Access memory 305
  12.3.1 Organization of Random-Access Memory 305
  12.3.2 Internal Construction of Random-Access Memory 306
  12.3.3 Expansion of RAM Size 308
  12.3.4 Available RAM ICs 309
12.4 Read-Only memory 310
  12.4.1 Organization of Read-Only Memory 310
  12.4.2 Internal Construction of Read-Only Memory 311
  12.4.3 Available EPROM ICs 312
12.5 Programmable Logic Devices 313
12.6 Programmable Read-Only Memory as Programmable Logic Device 315
  12.6.1 Architecture of Programmable Read-Only Memory 315
  12.6.2 Boolean Function Implementation with Programmable Read-Only Memory 315
12.7 Programmable Array Logic 316
  12.7.1 Architecture of Programmable Array Logic 316
  12.7.2 Boolean Function Implementation with Programmable Array Logic 318
  12.7.3 Available PAL Devices 320
  12.8 Programmable Logic Array 321
  12.8.1 Architecture of Programmable Logic Array 321
  12.8.2 Boolean Function Implementation with Programmable Logic Array 323
  12.8.3 Available PLA Devices 326
  12.8.4 PLA Based Logic Implementations 326
Exercises 330
Further Readings 332

13 REGISTER TRANSFER LOGIC 333
13.1 Introduction 333
13.2 Basics of Register Transfer Logic 333
13.3 Algorithmic State Machines 335
  13.3.1 The ASM Chart 335

13.3.2 Timing Consideration 338

13.4 Binary Multiplier 338

13.4.1 Datapath 339

13.4.2 ASM Chart 340

13.4.3 Control Logic 342

13.5 Binary Divider 348

13.5.1 Datapath 349

13.5.2 ASM Chart 350

13.5.3 Control Logic 352

Exercises 353

Further Readings 356

14 VERILOG HARDWARE DESCRIPTION LANGUAGE 357

14.1 Introduction 357

14.2 Overview of Verilog Language 357

14.2.1 Structural Description 358

14.2.2 Simulating the Description 359

14.2.3 Creating Ports of the Module 362

14.2.4 Creating a Testbench for a Module 363

14.3 Behavioral Modeling of Combinational Circuits 366

14.3.1 Procedural Models 366

14.3.2 Rules for Synthesizing Combinational Circuits 367

14.3.3 Combinational Logic Description Using Continuous Assign 368

14.4 Procedural Statements to Specify Combinational Logic 369

14.4.1 The Basics 369

14.4.2 Using Case Statements 370

14.4.3 Specifying Don't Care Situations 372

14.4.4 Procedural Loop Constructs 372

14.5 Procedural Modeling of Clocked Sequential Circuits 373

14.5.1 Modeling Finite State Machines 374

14.5.2 Rules for Synthesizing Sequential Systems 376

14.6 Explicit Style of Describing Finite State Machines 377

14.7 Hierarchical Description of Large System 379

14.7.1 The Clock Generator 379

14.7.2 The BCD Counter 380

14.7.3 The BCD to 7-Segment Display Decoder 381

14.7.4 The Top-level Module–Tying the Whole Circuit Together 381

14.8 Tying Behavioral and Structural Models Together 385

Exercises 386

Further Readings 389

15 LABORATORY EXPERIMENTS 390

15.0 Introduction 390

15.1 Binary and BCD Numbers 393

15.2 Digital Logic Gates 394

15.3 EXOR Gate Implementation 395

15.4 Simplification and Implementation of Boolean Function 396

15.5 Design and Implementation of a Combinational Circuit 397

15.6 Binary Adders and Subtractors 398

15.7 Decoder and Its Use in Combinational Logic Implementation 400

15.8 Code Converters 402

15.9 Multiplexer and Its Use in Combinational Logic Implementation 404

15.10 Flip-Flops 406

15.11 Design and Implementation of a Synchronous Sequential Circuit 408

15.12 Registers 409

15.13 Counters 411

15.14 Memory Unit 413

15.15 Verilog Simulation of a 4-bit Parallel Adder 415

15.16 Verilog Simulation of an Up/Down BCD Counter 416

PREFACE

Digital logic design is concerned with the interconnection among digital components and modules and is the term used to denote the design and analysis of digital systems. Digital circuits are employed in the design of systems such as digital computers, control systems, data communications, and many other applications that require electronic digital hardware. The design of digital systems requires many levels of skills. The low-level design skills are concerned with the concepts of digital electronics. The mid-level design skills require the knowledge of techniques and procedures of logical design using discrete components. The high-level design skills deal with the register transfer and architecture level design concepts. As the register transfer and architecture level circuit complexity is increasing day by day, manual design of such systems has become absolutely impossible. To overcome the difficulties, computer aided systems for modeling, simulating, and synthesizing digital systems have come into real practice.

In this book, basic techniques and procedures of logical level design of digital systems are discussed. The techniques are presented from the classical point of view. Boolean algebra and truth tables are used for the analysis and design of combinational circuits and state transition techniques for the analysis and design of sequential circuits. Register transfer logic design and digital systems modeling and simulation using Verilog hardware description language are also emphasized. The subject of the book is also known by other names such as logic design, digital logic, digital design, and switching circuits.

In this book, a balanced treatment has been made. For presenting the materials of this book, topics of digital electronics have been deliberately omitted and the topics are so presented that no such knowledge is needed. However, knowledge of basic digital electronics will obviously help understand the topics. On the other side, emphasis has been given towards practice than theory. Nowadays, a number of digital functions are available in ICs and digital systems can be designed using these ICs. The design techniques of these functions using discrete components are first discussed and then the commercially available ICs are introduced. This approach will help understand these functions and one would be able to use these functions for designing digital systems using these ICs. Most of the basic functions are designed using classical design techniques. However, to make the design procedure simple, many of the medium scale integration (MSI) level functions are designed using non-classical techniques suitable for the specific functions. A digital system is practically a sequential circuit. Designing a large sequential circuit using the classical state transition technique is very difficult. Register transfer logic level design methods of such systems are also emphasized. Moreover, to keep the topics balanced, computer aided method of modeling and simulation of digital systems using Verilog hardware description language is also introduced.

The book has been systematically organized and the presentation has been kept at a level suitable for a student with or without a basic knowledge of digital electronics. Every chapter includes a set of exercises. At the end of each chapter, a list of further readings is added. The students may also see these books as references.

The components used to construct digital systems are manufactured in integrated circuit form. Integrated circuits contain a large amount of interconnected digital circuits within a single small package. Medium scale integration (MSI) devices provide digital functions. It is very important for the logic designer to be familiar with the various digital components encountered in the integrated circuit form. For this reason, many MSI circuits are introduced throughout the book and their logical function explained. The various digital components that are represented are similar to commercial circuits. Classical design of these circuits is first introduced and then commercially available ICs are listed and some of them are discussed.

This book is suitable for a course in digital design for undergraduate programs of computer science, computer engineering, electrical engineering, communications engineering and electronics disciplines. Students of other disciplines taking courses on digital systems may also be benefited from this book.

Md. Mozammel Huq Azad Khan

ORGANIZATION OF THE BOOK

This book has been written as a textbook for courses in digital logic design for undergraduate programs of computer science, computer engineering, electrical engineering, communications engineering and electronics disciplines. Students of other disciplines taking courses on digital systems may also be benefited from this book.

The book has been systematically organized and the presentation has been kept at a level suitable for a student with or without a basic knowledge of digital electronics. The following is the brief description of the subjects that are covered in each chapter.

Chapter1 introduces the concepts of digital systems. Binary logic is informally introduced here from an intuitive point of view before proceeding with a formal definition of Boolean algebra.

Chapter 2 presents various binary systems suitable for representing information in digital components. The binary number system is explained and binary codes are discussed to show the representation of decimal and alphanumeric information.

Chapter 3 covers the Boolean algebra in details. The basic postulates and theorems of Boolean algebra are introduced and the correlation between Boolean expression and their corresponding logic diagram is also discussed.

Chapter 4 presents the digital logic gates. All possible logic operations for two variables are investigated and from that the most useful logic gates used in the design of digital systems are determined. The characteristics of integrated circuit gates are mentioned.

Chapter 5 covers the minimization of Boolean functions. Algebraic method of simplification of Boolean functions is introduced followed by detail discussion of K-map method. Then two algorithmic procedure of simplification of Boolean functions, namely Quine-McCluskey method and iterated consensus method, are introduced.

Chapter 6 outlines the formal procedures for the design and analysis of combinational circuits. Besides classical two-level AND-OR and OR-AND circuits, other two-level circuits such as NAND-NAND and NOR-NOR circuits are introduced. Multi-level NAND and NOR circuits are also discussed. Finally, two-level AND-EXOR and three-level AND-OR-EXOR circuits are introduced.

Chapter 7 presents the design of arithmetic and magnitude comparator circuits. Basic arithmetic circuits such as half-adder, full-adder, half-subtractor, and full-subtractor are designed using classical design techniques. Then the more complicated functions are designed using these basic functions. Commercially available circuits are also introduced.

Chapter 8 covers the design of various components used in the design of digital systems, such as encoders, decoders, code converters, multiplexers, and demultiplexers. Commercially available ICs are also introduced. The use of decoder, multiplexer, and demultiplexer in the design of combinational circuits is also discussed.

Chapter 9 introduces the construction and behavior of various types of flip-flops as building blocks for sequential circuits. Applications of flip-flops are also discussed.

Chapter 10 outlines the formal procedure for the design and analysis of clocked sequential circuits. The chapter starts by presenting various methods for representing sequential circuits such as state diagram, state table, and state equations. These methods are shown to be convenient tools for analyzing and designing sequential circuits.

Chapter 11 presents various sequential digital components such as registers and counters. These digital components are the basic building blocks from which more complex digital systems are constructed. Design of these components using discrete gate is first discussed and then commercially available ICs are introduced. Application of these sequential digital components are also introduced.

Chapter 12 introduces the memory units and programmable logic devices (PLDs). Random-access memory (RAM) and read-only memory (ROM) are discussed. Three types of PLDs such as programmable read-only memory (PROM), programmable array logic (PAL), and programmable logic array (PLA) and their use in the design of digital circuits are discussed. These three PLD devices are extensively used in the design and implementation of complex digital circuits.

Chapter 13 discusses register transfer logic (RTL) level design of digital systems. Practically all digital systems are sequential circuits. Designing very large sequential circuits using the classical state transition techniques is very difficult. These large systems are designed as RTL circuits. Basic foundation of RTL design is introduced and the method is illustrated by designing two practically important circuits like binary multiplier and divider.

Chapter 14 introduces the Verilog hardware description language in a tutorial fashion. As design complexity of RTL level circuits is becoming very high, computer-aided modeling, simulation, and synthesis have become very important. The Verilog is one of the most popular hardware description languages, which supports description of both combinational and sequential circuits using either structural model or behavioral model. The language also allows simulation and synthesis of the descriptions with suitable simulator and synthesis tools. The most important feature of the Verilog language is that its rules and syntax are very similar to C programming language.

Chapter 15 outlines 14 experiments that can be performed in the laboratory with hardware that is readily and inexpensively available commercially. These experiments use standard integrated circuits of the TTL type. Each experiment is presented in a step-by-step fashion so that the students can perform the experiment easily in the laboratory. Each experiment is divided into two parts – pre-lab work and lab work. The pre-lab work helps the students prepare themselves with the detail circuit diagram and procedure for checking the operation of the circuit. The lab works guide the students to perform the experiment using the pre-lab work methods. Doing the experiments in this chapter will enhance the knowledge of practical application of digital logic design. The chapter also outlines 2 Verilog simulation experiments, which will help student practically learn modeling and simulation of very large digital systems using the Verilog hardware description language.

LESSON PLAN

This book is written as a textbook for courses in Digital Logic Design for undergraduate programs of computer science, computer engineering, electrical engineering, communications engineering and electronics disciplines. If the undergraduate program has two courses on Digital Logic Design, then the full texts of the book can be covered. In the first course, emphasis can be given on Boolean algebra and combinational logic design. Chapter 1 through 8 can be covered for this purpose. In the second course, emphasis can be given on sequential circuit design, circuit design with PLDs, RTL level design of digital systems, and Verilog modeling and simulation of large digital systems. For this purpose Chapters 9 through 14 can be taught. The experiments of Chapter 15 can be divided into two groups depending on the needs and availability of resources in the laboratory.

If the undergraduate program has only one course on Digital Logic Design, then a careful selection of the topics should be made. Instructors can do this from their own choice and depending on the requirement of the program. However, a guideline for selecting such topics may be as follows, which the author follows in his own course for the students of Computer Science and Engineering:

1. Chapter 1 (Introduction to Digital Systems) may be skipped by simply giving an introduction to digital circuit.
2. Chapter 2 (Number Systems and Codes) may be skipped if the students already have knowledge of binary number and arithmetic. Otherwise, only Sections 2.4, 2.5, and 2.8 can be taught.
3. Chapter 3 (Boolean Algebra) has to be covered in depth.
4. Chapter 4 (Logic Gates) can be skipped by simply introducing the digital gates and their truth tables. If time permits, then Section 4.4 and 4.6 can be covered.
5. Chapter 5 (Simplification of Boolean Functions) can be taught partially by giving emphasis on only K-map and Quine-McCluskey methods of simplification. For this purpose, Sections 5.3, 5.4, and 5.5 can be taught.
6. Chapter 6 (Combinational Logic Design) can be covered partially by focusing on only two-level circuit design and analysis. For this purpose, Sections 6.2, 6.3, 6.4, 6.5, and 6.8 can be covered.
7. Chapter 7 (Arithmetic and Comparator Circuits) may also be taught partially by giving emphasis on only binary arithmetic and comparator circuits. For this purpose, Sections 7.2, 7.4, 7.5, 7.6, and 7.9 can be covered.
8. Chapter 8 (Encoders, Decoders, Multiplexers, and Demultiplexers) can be taught in full.
9. Chapter 9 (Flip-Flops) has to be taught in full.
10. Chapter 10 (Sequential Logic Design) has to be covered in full.
11. Chapter 11 (Registers and Counters) can be taught partially by covering the Sections 11.2, 11.3, 11.4, 11.5, 11.6, 11.7, 11.9, 11.10, 11.11 (except 11.11.5), 11.12, and 11.13.

12. Chapter 12 (Memories and Programmable Logic Devices) can be covered partially focusing on only circuit design with PLDs. For this purpose, Sections 12.5, 12.6, 12.7, and 12.8 may be covered.
13. Chapter 13 (Register Transfer Logic) may be skipped.
14. Chapter 14 (Verilog Hardware Description Language) may be touched, if time permits, by teaching only Section 14.2.
15. Laboratory experiments from Chapter 15 (Laboratory Experiments) may be judiciously coined depending on the instructor's choice and availability of laboratory resources.

CHAPTER 1

INTRODUCTION TO DIGITAL SYSTEMS

1.1 Introduction

When most of us hear the term *digital*, we immediately think of "digital calculator" or "digital computer". However, it should be noted that calculators and computers represent only one of the many applications of digital circuits and principles. Digital circuits are used in electronic products such as video games, microwave ovens, and automobile control systems, and in test equipment such as meters, generators, and oscilloscopes. Digital techniques have also replaced a lot of the older "analog circuits" used in consumer products such as radios, TV sets, and high fidelity sound recording and playback equipment.

In this book we are going to study the principles and techniques that are common to all digital systems.

1.2 Digital and Analog Systems

In most of our daily endeavors, we are constantly dealing with *quantities*. Quantities are measured, monitored, recorded, manipulated arithmetically, observed, or in some other way utilized in most physical systems. There are basically two ways of representing the numerical value of quantities: *analog* and *digital*.

In *analog representation* a quantity is represented by a voltage, current, or meter movement that is proportional to the value of that quantity. An example is an automobile speedometer, in which the deflection of the needle is proportional to the speed of the auto. The angular position of the needle represents the value of the auto's speed, and the needle follows any changes that occur as the auto speeds up or slows down. Analog quantities have an important characteristic: *they can vary over a continuous range of values*. The automobile speed can have any value between zero and, say, 100 mph.

In *digital representation* the quantities are represented not by proportional quantities but by symbols called *digits*. As an example, consider the digital watch, which provides the time of day in the form of decimal digits that represent hours, minutes, and seconds. As we know, the time of day changes continuously, but the digital watch reading does not change continuously; rather it changes in steps of one per second. In other words, this digital representation of the time of day changes in *discrete* steps.

A *digital system* is a combination of devices designed to manipulate physical quantities or information that are represented in digital form; that is, they can take on only discrete values. These devices are most often electronic, but they can be mechanical, magnetic, or pneumatic. Some of the more familiar digital systems include digital computers and calculators, digital audio and video equipment, and the digital telephone system.

An *analog system* contains devices that manipulate physical quantities that are represented in analog form. In an analog system, the quantities can vary over a continuous range of values. Some common analog systems are audio amplifier, magnetic tape recording and playback equipment, and the automobile odometer.

Advantages of Digital Techniques

An increasing majority of applications in electronics, as well as in most other technologies, use digital techniques to perform operations that were once performed using analog methods. The chief reasons for the shift to digital technology are:

1. Digital systems are generally easier to design.
2. Information storage is easy.
3. Accuracy and precision are greater.
4. Operation can be programmed; that is, operation can be controlled by a set of stored instructions.
5. Digital circuits are less affected by noise.
6. More digital circuitry can be fabricated on IC chips.

1.3 Binary Numbers in Digital Systems

In digital systems, information is represented and manipulated in digital form. Most of the digital devices operate in only two stable states and are inherently *binary* in nature. The two stable states of a digital device can be used for representing only two digits. Fortunately, the binary number system has only two digits: 0 and 1, and any number can be represented by using only these two digits. Therefore, *binary number system* is used for representing and manipulating information in digital systems. We shall discuss binary number system in detail in Chapter 2.

In digital systems, binary digits can be represented by any device that has only two operating states or possible conditions. For example, a switch has only two states: open and closed. We can arbitrarily let an open switch represent binary 0 and a closed switch represent binary 1. With this assignment we can now represent any binary number as illustrated in Figure 1.1, where the states of the various switches represent binary number 10010.

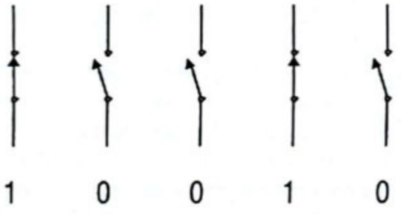
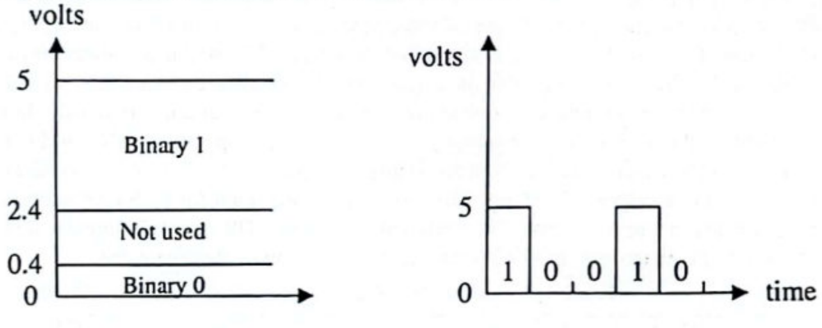


Figure 1.1 Using switches to represent binary numbers.

There are numerous other devices which have only two operating states or can be operated in two extreme conditions, among these are: light bulbs (bright or dark), diode (conducting or nonconducting), relay (energized or deenergized), transistor (cut off or saturated), and magnetic tape (magnetized or demagnetized).

In electronic digital systems, binary digits are represented by voltages that are present at the inputs and outputs of the various circuits. Typically, the binary 0 and binary 1 are represented by two nominal voltage levels. For example, zero volts (0V) might represent binary 0, and +5V might represent binary 1. In actuality, because of circuit variations, the binary 0 and binary 1 would be represented by voltage ranges. This is illustrated in Figure 1.2(a), where any voltage between 0 and 0.4V represent a binary 0 and any voltage between 2.4 and 5V represent a binary 1. All input and output signals will normally fall within one of these ranges except during transition from one level to another. Figure 1.2(b) shows a typical digital signal as it sequences through the binary value 10010.



(a) (b)

Figure 1.2 (a) Typical voltage assignments in digital system, (b) typical digital signal.

1.4 Binary Logic

Binary logic deals with variables that take on two discrete values and with operations that assume logical meaning. The two values the variables take may be called by different names (e.g., *true* and *false*, *yes* and *no*, etc.), but in the digital system it is convenient to think in terms of binary digits and assign the values of 1 and 0. Binary logic is used to describe, in a mathematical way, the manipulation and processing of binary information. It is particularly suited for the analysis and design of digital systems. The binary logic to be introduced here is equivalent to an algebra called *Boolean algebra*. We shall discuss Boolean algebra in detail in Chapter 3.

1.4.1 Definition of Binary Logic

Binary logic consists of binary variables and logical operations. The variables are designated by letters of the alphabet such as A, B, C, x, y, z, etc., with each variable having two and only two distinct possible values: 0 and 1. There are three basic logical operations: AND, OR, and NOT.

1. **AND:** This operation is represented by a dot or by the absence of an operator. For example, $x \cdot y = z$ or $x\,y = z$ is read "x AND y is equal to z". The logical operation AND is interpreted to mean that $z = 1$ if and only if $x = 1$ and $y = 1$; otherwise $z = 0$.
2. **OR:** This operation is represented by a plus sign. For example, $x + y = z$ is read "x OR y is equal to z", meaning that $z = 1$ if $x = 1$ or $y = 1$ or if both $x = 1$ and $y = 1$. If both $x = 0$ and $y = 0$, then $z = 0$.

3. **NOT:** This operation is represented by prime (sometimes by a bar). For example, $x' = z$ (or $\bar{x} = z$) is read "x not is equal to z", meaning that z is what x is not. In other words, if $x = 1$, then $z = 0$; but if $x = 0$, then $z = 1$.

1.4.2 Truth Table

For each combination of the values of x and y, there is a value of z specified by the definition of the logical operation. These definitions may be listed in a compact form using truth tables. A *truth table* is a table of all possible combinations of the variables showing the relation between the values that the variables may take and the result of the operation. For example, the truth tables for operations AND and OR with variables x and y are obtained by listing all possible values that the variables may have when combined in pairs. The result of the operation for each combination is then listed in a separate row. The truth tables for AND, OR, and NOT are listed in Table 1.1. These tables clearly demonstrate the definition of the operations.

Table 1.1 Truth tables of logical operations.

AND	
$x\ y$	$z = x \cdot y$
0 0	0
0 1	0
1 0	0
1 1	1

OR	
$x\ y$	$z = x + y$
0 0	0
0 1	1
1 0	1
1 1	1

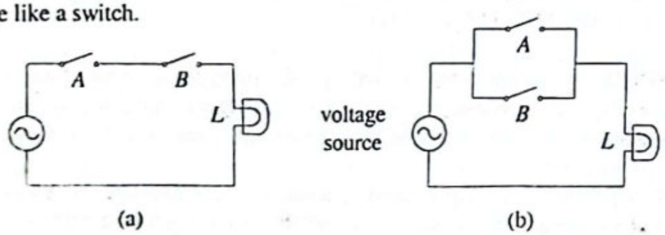
NOT	
x	$z = x'$
0	1
1	0

1.4.3 Switching Circuits

The use of binary variables and the application of binary logic are demonstrated by the simple switching circuits of Figure 1.3. Let the switches A and B represent two binary variables with values equal to 0 when the switch is open and 1 when the switch is closed. Similarly, let the lamp L represent a third binary variable equal to 1 when the light is on and 0 when off. For the switches in series, the light turns on if A and B are closed. For the switches in parallel, the light turns on if A or B is closed. It is obvious that the two circuits can be expressed by means of binary logic with the AND and OR operations, respectively;

$L = A \cdot B$ for the circuit of Figure 1.3(a),
$L = A + B$ for the circuit of Figure 1.3(b).

Electronic digital circuits are sometimes called *switching circuits* because they behave like a switch.

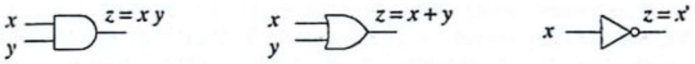


(a) (b)

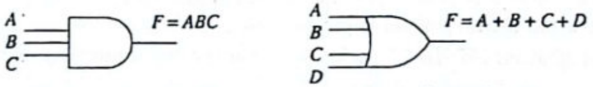
Figure 1.3 Switching circuits that demonstrate binary logic – (a) switches in series – logic AND. (b) switches in parallel – logic OR.

1.4.4 Logic Gates

Electronic digital circuits are also called *logic circuits* because, with the proper input, they establish logical manipulation paths. Logic circuits that perform the logic operations of AND, OR, and NOT are shown with their symbols in Figure 1.4. These circuits, called *logic gates*, are building blocks of digital systems that produce a 0 or 1 output if input logic requirements are satisfied. The NOT gate is sometimes called an *inverter* since it inverts a binary signal.



(a) Two-input AND gate (b) Two-input OR gate (c) NOT gate or inverter



(d) Three-input AND gate (e) Four-input OR gate

Figure 1.4 Symbols for digital logic gates.

The input signals x and y in the two-input gates of Figure 1.4 may exist in one of four possible states: 00, 01, 10, or 11. These input signals are shown in Figure 1.5, together with the output signals for the AND and OR gates. The timing diagrams in Figure 1.5 illustrate the response of each gate to each of the four possible input binary combinations. The response of NOT gate for input x is also shown in the figure.

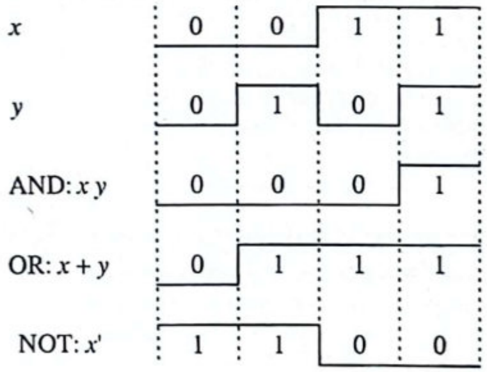


Figure 1.5 Input-output signals for AND, OR, and NOT gates.

AND and OR gates may have more than two inputs. An AND gate with three inputs and an OR gate with four inputs are shown in Figure 1.4. The three-input AND gate responds with an 1 output if all three input signals are 1. The output produces a 0 signal if any input is 0. The four-input OR gate responds with an 1 when any input is an 1. Its output becomes 0 if all input signals are 0. We shall discuss logic gates in detail in Chapter 4.

1.5 Integrated Circuits

Digital circuits are invariably constructed with integrated circuits. An integrated circuit (abbreviated as IC) is a small silicon semiconductor crystal, called a *chip*, containing electrical components such as transistors, diodes, resistors, and capacitors. The various components are interconnected inside the chip to form an electronic circuit. The chip is mounted on a metal or plastic package, and connections are welded to external pins to form the IC. Digital systems are constructed with ICs.

ICs are classified as follows, depending on the number of gates they contain:

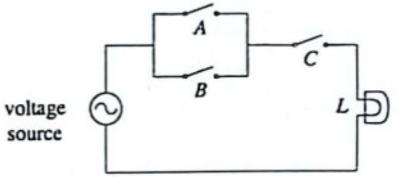
- **Small-scale integration (SSI):** Several logic gates in a single IC.
- **Medium-scale integration (MSI):** IC perform a complete logic function with 10 to 100 gates.
- **Large-scale integration (LSI):** IC performs a complete logic function with more than 100 gates.
- **Very-large-scale integration (VLSI):** IC performs complex logic operation with thousands of gates.

Exercises

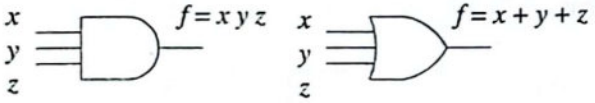
1.1 Briefly describe the major difference between analog and digital quantities.

1.2 What are the advantages of digital techniques over analog?

1.3 Express the following switching circuit in binary logic notation.



1.4 Show the output responses of the following gates using timing diagrams.



Further Reading

1. Richard S. Sandige, *Modern Digital Design*, McGraw-Hill Publishing Company, 1990.
2. Ronald J. Tocci and Neal S. Widmer, *Digital Systems: Principles and Applications*, 8th ed., Prentice-Hall, Inc., 2001.
3. V. K. Jain, *Switching Theory and Digital Electronics*, Khanna Publishers, India, 2002.
4. M. Morris Mano, *Digital design*, 3rd ed., Pearson Education, 2002.

CHAPTER 2
NUMBER SYSTEMS AND CODES

2.1 Introduction

We all are familiar with the number system in which an ordered set of ten symbols – 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, known as *digits* – are used to specify any number. This number system is popularly known as the *decimal number system*. The *radix* or *base* of this number system is 10 (number of distinct digits). Any number is a collection of these digits. For example, 2356.3294 signifies a number with an *integer part* equal to 2356 and a *fractional part* equal to .3294, separated from the integer part with a *radix point* (.), also known as *decimal point*. There are some other systems also used to represent numbers. Some of the other commonly used number systems are *binary*, *octal*, and *hexadecimal* number systems. These number systems are widely used in digital systems and, therefore, the knowledge of these number systems is very essential for understanding, analyzing and designing digital systems.

Digital systems are required to handle data, which may be numerals, alphabets, or special characters. But digital systems use binary signals for their operations. Therefore, the information available in any other form is required to be converted into suitable binary form before it can be processed by digital circuits. This means that the information available in the form of numerals, alphabets, and special characters or in any combination of these must be converted into binary format. To achieve this, a process of coding is employed whereby each numeral, alphabet, or special character is coded in a unique combination of 0s and 1s using a coding scheme, known as a *code*. The process of coding is known as *encoding*.

There can be a variety of coding schemes (codes) to serve different purposes, such as arithmetic operations, data entry, error detection and correction, etc. In digital systems, a large number of codes are in use. Selection of a particular code depends on its suitability for the purpose. In one digital system, different codes may be used for different operations and it may be necessary to convert data from one code to another code.

2.2 Number Systems

In general, in any number system there is an ordered set of symbols known as *digits* with rules defined for performing arithmetic operations like addition, multiplication, etc. A collection of these digits makes a number which in general has two parts – *integer* and *fractional*, set apart by a *radix point* (.), that is

$$(N)_r = \underbrace{d_{n-1}d_{n-2}\cdots d_i\cdots d_1 d_0}_{\text{Integer portion}}\underset{\underset{\text{Radix point}}{\uparrow}}{.}\underbrace{d_{-1}d_{-2}\cdots d_{-f}\cdots d_{-m}}_{\text{Fractional portion}}$$

where N = a number
r = radix or base of the number system
n = number of digits in integer portion

m = number of digits in fractional portion

d_{n-1} = most significant digit (MSD)

d_{-m} = least significant digit (LSD)

and $0 \le (d_i \text{ or } d_{-f}) \le (r-1)$.

To distinguish between numerals of different bases, we enclose the digits of the number in parentheses and write a subscript equal to the base used (except sometimes for decimal numerals, where the context makes it obvious that it is decimal).

The digits in a number are placed side by side and each position in the number is assigned a *weight* of r^i or r^{-f}. Thus, the quantity represented by the number is

$$\begin{aligned}(N)_r &= d_{n-1}r^{n-1} + d_{n-2}r^{n-2} + \cdots + d_i r^i + \cdots + d_1 r^1 + d_0 r^0 \\ &\quad + d_{-1}r^{-1} + d_{-2}r^{-2} + \cdots + d_{-f}r^{-f} + \cdots + d_{-m}r^{-m} \\ &= \sum_{i=-m}^{n-1} d_i r^i\end{aligned}$$

For example, the value of the decimal number 2356.3294 is

$$\begin{aligned}(2356.3294)_{10} &= 2\cdot10^3 + 3\cdot10^2 + 5\cdot10^1 + 6\cdot10^0 + 3\cdot10^{-1} + 2\cdot10^{-2} + 9\cdot10^{-3} + 4\cdot10^{-4} \\ &= 2000 + 300 + 50 + 6 + 0.3 + 0.02 + 0.009 + 0.0004 \\ &= 2356.3294\end{aligned}$$

As each position of the number is assigned a weight, this number system is called *positional number system.*

2.3 Complements

Complements are used for simplifying the subtraction operation. There are two types of complements for each base-r system:

(1) the r's complement and
(2) the $(r-1)$'s complement.

When the value of the base is substituted, the two types receive the names 2's and 1's complement for binary numbers, or 10's and 9's complement for decimal numbers.

2.3.1 The r's Complement

Given a positive number N in base r with an integer part of n digits, the r's complement of N is defined as

$$\text{The } r\text{'s complement of } N = \begin{cases} r^n - N & \text{if } N \ne 0 \\ 0 & \text{if } N = 0 \end{cases}$$

The following numerical examples will help clarify the definition.

The 10's complement of 52520 = $10^5 - 52520 = 47480$.

[The number of digits in the integer part of the number is $n = 5$.]

The 10's complement of .3267 = $10^0 - 0.3267 = 0.6733$.
[The number of digits in the integer part of the number is $n = 0$.]

The 10's complement of 25.639 = $10^2 - 25.639 = 74.361$.

From the definition and the examples, it is clear that *the 10's complement of a decimal number can be formed by leaving all least significant zeros unchanged, subtracting the first nonzero least significant digit from 10, and then subtracting all other higher significant digits from 9.*

2.3.2 The $(r-1)$'s Complement

Given a positive number N in base r with an integer part of n digits and a fractional part of m digits, the $(r-1)$'s complement of N is defined as

The $(r-1)$'s complement of $N = r^n - r^{-m} - N$.

Some numerical examples follow:

The 9's complement of 52520 = $10^5 - 10^{-0} - 52520 = 100000 - 1 - 52520 = 99999 - 52520 = 47479$. [No fractional part, so $m = 0$.]
The 9's complement of .3267 = $10^0 - 10^{-4} - .3267 = 1 - .0001 - .3267 = .9999 - .3267 = .6732$. [No integer part, so $n = 0$.]
The 9's complement of 25.639 = $10^2 - 10^{-3} - 25.639 = 100 - .001 - 25.639 = 99.999 - 25.639 = 74.360$.

From the examples, we see that *the 9's complement of a decimal number is formed simply by subtracting every digit from 9*. Since the $(r-1)$'s complement is very easily obtained, it is sometimes convenient to use it when the r's complement is desired. From the definition and from a comparison of the results obtained in the examples, it follows that *the r's complement can be obtained from the (r – 1)'s complement after the addition of* r^{-m}. For example, the 10's complement of 25.639 is 74.361 and this can be obtained by adding $10^{-3} = 0.001$ to the 9's complement of 25.639, that is 74.360, to give 74.361.

It is worth mentioning that the complement of the complement restores the number to its original value. The r's complement of N is $r^n - N$ and r's complement of $(r^n - N)$ is $r^n - (r^n - N) = N$; similarly for the $(r-1)$'s complement.

2.3.3 Subtraction with r's Complement

The direct method of subtraction uses the borrow concept. In this method, we borrow a 1 from a higher significant position when the minuend digit is smaller than the corresponding subtrahend digit. This seems to be easiest when people perform subtraction with paper and pencil. When subtraction is implemented by means of digital components, this method is found to be less efficient than the method that uses complements and addition as stated below.

The subtraction of two positive numbers $(M - N)$, both of base r, may be done as follows:

1. Find the r's complement of the subtrahend N and add it to the minuend M.
2. Inspect the result obtained in step 1 for an end carry.
   (a) If an end carry occurs, the result is a positive number. Discard the end carry to find the magnitude of the result.
   (b) If an end carry does not occur, the result is a negative number. Take the r's complement of the number obtained in step 1 to find the magnitude of the result.

The following examples illustrate the procedure:

Example 1:

$$\begin{array}{rcl} 72532 & \rightarrow & 72532 \\ (-)\ 3250 & \rightarrow & (+)\ 96750 \text{ (10's complement of 3250)} \\ \hline 69282 & & (1)\ 69282 \\ & & \downarrow \\ & & \text{end carry} \end{array}$$

The result is 69282.

Example 2:

$$\begin{array}{rcl} 3250 & \rightarrow & 03250 \\ (-)\ 72532 & \rightarrow & (+)\ 27468 \text{ (10's complement of 72532)} \\ \hline -\ 69282 & & 30718 \text{ (no end carry)} \end{array}$$

The result is – 69282 (10's complement of 30718)

The proof of the procedure is: The r's complement of N is $(r^n - N)$. Adding it to M gives $(M + r^n - N)$. For numbers having an integer part of n digits, r^n is equal to a 1 in the $(n+1)$st position (what has been called the "*end carry*"). Since both M and N are assumed to be positive, then

(a) $(M + r^n - N) \geq r^n$ if $M \geq N$, or

(b) $(M + r^n - N) < r^n$ if $M < N$.

In case (a) the result is positive and equal to $M - N$, which is directly obtained by discarding the end carry r^n. In case (b) the result is negative and equal to $-(N - M)$. This case is detected from the absence of an end carry. The result is obtained by taking a second r's complement and adding a negative sign:

$$-[r^n - (M + r^n - N)] = -(N - M).$$

2.3.4 Subtraction with (r - 1)'s Complement

The procedure for subtraction with $(r-1)$'s complement is exactly the same as the one used with the r's complement except for one variation, called "*end-around carry*", as shown below. The subtraction of $M - N$, both positive numbers in base r, may be calculated in the following manner:

1. Find the $(r-1)$'s complement of the subtrahend N and add it to the minuend M.
2. Inspect the result obtained in step 1 for an end carry.

(a) If an end carry occurs, the result is a positive number. Add the end carry to the least significant digit (end-around carry) to find the magnitude of the result.

(b) If an end carry does not occur, the result is a negative number. Take the $(r-1)$'s complement of the number obtained in step 1 to find the magnitude of the result.

The following examples illustrate the procedure:

Example 1:

$$\begin{array}{rcl} 72532 & \rightarrow & 72532 \\ (-)\ 3250 & \rightarrow & (+)\ 96749 \text{ (9's complement of 03250)} \\ \hline 69282 & & (1)\ 69281 \\ & & \hookrightarrow (+)\ 1 \text{ end-around carry} \\ \hline & & 69282 \end{array}$$

The result is 69282.

Example 2:

$$\begin{array}{rcl} 3250 & \rightarrow & 03250 \\ (-)\ 72532 & \rightarrow & (+)\ 27467 \text{ (9's complement of 72532)} \\ \hline -\ 69282 & & 30717 \text{ (no end carry)} \end{array}$$

The result is – 69282 (9's complement of 30717).

The proof of the procedure is: The $(r-1)$'s complement of N is $(r^n - r^{-m} - N)$. Adding it to M gives $(M + r^n - r^{-m} - N)$. For numbers having an integer part of n digits, r^n is equal to a 1 in the $(n+1)$st position. Since both M and N are assumed to be positive, then:

(a) $(M + r^n - r^{-m} - N) \geq r^n - r^{-m}$ if $M \geq N$, or

(b) $(M + r^n - r^{-m} - N) < r^n - r^{-m}$ if $M < N$.

For numbers having a fractional part of m digits, subtracting r^{-m} is equal to subtracting a 1 (in base r) from the least significant digit ($-m$th digit). In case (a) the result is positive and equal to $M - N$, which is directly obtained by discarding the end carry r^n and adding a 1 to the least significant digit. In case (b) the result is negative and equal to $-(N - M)$. This case is detected from the absence of an end carry. The result is obtained by taking a second $(r-1)$'s complement and adding a negative sign:

$$-[r^n - r^{-m} - (M + r^n - r^{-m} - N)] = -(N - M).$$

2.4 Binary Number System

2.4.1 Binary Number Representation

The number system with base (or radix) two is known as the *binary number system*. Only two digits are used to represent numbers in this system and these are 0 and 1. These are known as *bits* (bit is the abbreviation of *bi*nary digi*t*). It is a positional number system and every position is assigned a weight of 2^i.

Table 2.1 illustrates counting in binary number system. The corresponding decimal numbers are given in the right-hand column. The left-most bit is known as the *most significant bit (MSB)* and the right-most bit is known as the *least significant bit (LSB)*. Any number of 0s can be added to the left of the number without changing the value of the number. In the binary number system, a group of four bits is known as a *nibble*, and a group of eight bits is known as a *byte*.

In binary number system, using *n* bit, number ranging from 0 to ($2^n - 1$) can be represented. For example, using 4 bits, numbers ranging from 0 to 15 (that is $2^4 - 1$) can be represented as seen in Table 2.1.

2.4.2 Binary to Decimal Conversion

A binary number can be converted to decimal number by using the formula

$$(N)_2 = \sum_{i=-m}^{n} d_i 2^i$$

For example,

$$\begin{aligned}(1010.011)_2 &= 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 + 0\times2^{-1} + 1\times2^{-2} + 1\times2^{-3}\\ &= 8 + 0 + 2 + 0 + 0 + 0.25 + 0.125\\ &= (10.375)_{10}.\end{aligned}$$

Table 2.1 4-bit binary numbers and their corresponding decimal numbers.

Binary number	Decimal number
0000	00
0001	01
0010	02
0011	03
0100	04
0101	05
0110	06
0111	07
1000	08
1001	09
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

2.4.3 Decimal to Binary Conversion

Any decimal number can be converted into its equivalent binary number. This conversion is done separately for integer and fractional parts.

Conversion of Integer Part

Let $(N)_{10}$ be a decimal integer number whose binary equivalent is $b_{n-1}b_{n-2}\cdots b_1b_0$, where *b*s are bits. Therefore, we can write that

$$(N)_{10} = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_1 2^1 + b_0 2^0$$

If we divide $(N)_{10}$ by 2, we get

$$(N/2)_{10} = (b_{n-1}2^{n-2} + b_{n-2}2^{n-3} + \cdots + b_1 2^0) + b_0$$

where, $(b_{n-1}2^{n-2} + b_{n-2}2^{n-3} + \cdots + b_1 2^0)$ is the quotient and b_0 is the remainder. In this division, the least significant bit b_0 is separated out. By repeated division of the quotient by 2, we can separate out the higher significant bits. This process is explained below for $(13)_{10}$:

	Quotient	Remainder	
13/2	6	1	LSB
6/2	3	0	
3/2	1	1	
1/2	0	1	MSB

Thus $(13)_{10} = (1101)_2$.

Conversion of Fractional Part

Let $(N)_{10}$ be a decimal fractional number whose binary equivalent is $.b_{-1}b_{-2}\cdots b_{-m}$ and we can write that

$$(N)_{10} = b_{-1}2^{-1} + b_{-2}2^{-2} + \cdots + b_{-m}2^{-m}$$

If we multiply $(N)_{10}$ by 2, we get

$$(N\times 2)_{10} = b_{-1} + (b_{-2}2^{-1} + \cdots + b_{-m}2^{-(m-1)})$$

where, b_{-1} is the integer part and $(b_{-2}2^{-1} + \cdots + b_{-m}2^{-(m-1)})$ is the fraction part of $(N\times 2)_{10}$. In this multiplication, the most significant bit b_{-1} is separated out. By repeated multiplication of the fractional part, we can separate out the lower significant bits. This process is explained below for $(.6875)_{10}$:

$.6875 \times 2 = 1.3750 \longrightarrow 1$ MSB
$.3750 \times 2 = 0.7500 \longrightarrow 0$
$.7500 \times 2 = 1.5000 \longrightarrow 1$
$.5000 \times 2 = 1.0000 \longrightarrow 1$ LSB

Thus $(.6875)_{10} = (.1011)_2$.

The conversion of decimal numbers with both integer and fractional parts is done by converting the integer and fraction separately and then combining the two results together. Using the results of the above examples we have

$(13.6875)_{10} = (1101.1011)_2$.

2.4.4 2's Complement of Binary Numbers

Given a positive binary number N with an integer part of n bits, the 2's complement of N is defined as

$$\text{The 2's complement of } N = \begin{cases} 2^n - N & \text{if} \quad N \neq 0 \\ 0 & \text{if} \quad N = 0 \end{cases}$$

The following numerical examples will help clarify the definitions.

The 2's complement of $(101100)_2 = (2^6)_{10} - (101100)_2$
$= (1000000 - 101100)_2 = (010100)_2$.

The 2's complement of $(.0110)_2 = (2^0)_{10} - (.0110)_2$
$= (1 - .0110)_2 = (.1010)_2$.

From the definition and the examples, it is clear that *the 2's complement can be formed by leaving all least significant zeros and the first nonzero bit unchanged, and then replacing 1s by 0s and 0s by 1s in all other higher significant bits.*

2.4.5 1's Complement of Binary Numbers

Given a positive binary number N with an integer part of n bits and a fractional part of m bits, the 1's complement of N is defined as

$$\text{The 1's complement of } N = 2^n - 2^{-m} - N .$$

Some numerical examples follow:

The 1's complement of $(101100)_2 = (2^6 - 2^0)_{10} - (101100)_2$
$= (111111 - 101100)_2 = (010011)_2$. [No fractional part, so $m = 0$.]

The 1's complement of $(.0110)_2 = (2^0 - 2^{-4})_{10} - (.0110)_2$
$= (.1111 - .0110)_2 = (.1001)_2$.

From the examples, we see that *the 1's complement of a binary number is formed by changing 1s to 0s and 0s to 1s.* Since the 1's complement is very easily obtained, it is sometimes convenient to use it when the 2's complement is desired. From the definition and from a comparison of the results obtained in the examples, it follows that *the 2's complement can be obtained from the 1's complement after addition of a 1 to the least significant bit.* For example, the 2's complement of $(10110100)_2$ is $(01001100)_2$ and this can be obtained from the 1's complement $(01001011)_2$ by adding 1 to give $(01001100)_2$.

2.5 Binary Arithmetic

We all are familiar with the arithmetic operations such as addition, subtraction, multiplication, and division of decimal numbers. Similar operations can be performed on binary numbers.

2.5.1 Binary Addition

The rules of binary addition are given in Table 2.2.

Table 2.2 Rules of binary addition.

Augend	Addend	Result	Carry	Sum
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	10	1	0

In the first three rows in Table 2.2, the result is of one bit only and there is no *carry*, that is, carry is 0. In the fourth row, the result is of two bits and a *carry* is produced (since in binary two is 10), that is, carry is 1. Similar to decimal addition this carry is added to the next higher bit position. Binary addition is explained using examples below:

Example 1:
1 0 1 1 Augend
(+) 0 1 0 0 Addend
1 1 1 1 Sum

Example 2:
1 0 1 1
(+) 1 1 0 0
1 0 1 1 1
↳ Carry

Example 3:
(1) (1) ← Carry
0 1 0 1
(+) 1 1 0 1
1 0 0 1 0
↳ Carry

2.5.2 Binary Subtraction

The rules of binary subtraction are given in Table 2.3.

Table 2.3 Rules of binary subtraction.

Minuend	Subtrahend	Borrow	Difference
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

In Table 2.3, except in the second row, the borrow is 0. We borrow a 1 from a higher significant position when the minuend bit is smaller than the corresponding subtrahend bit. When the borrow is 1, as in the second row, this is to be subtracted from the next higher bit as it is done in decimal subtraction. Binary subtraction is explained using the examples below:

Example 1:

$$\begin{array}{rl} 1\,0\,1\,1 & \text{Minuend} \\ (-)\,1\,0\,1\,0 & \text{Subtrahend} \\ \hline 0\,0\,0\,1 & \text{Difference} \end{array}$$

Example 2:

$$\begin{array}{rl} (1) & \longleftarrow \text{Borrow} \\ 1\,0\,1\,1 & \\ (-)\,0\,1\,1\,0 & \\ \hline 0\,1\,0\,1 & \end{array}$$

2.5.3 Binary Multiplication

Binary multiplication is similar to decimal multiplication. In binary, each partial product is either zero (multiplication by 0) or exactly same as the multiplicand (multiplication by 1). An example of binary multiplication is given below:

$$\begin{array}{rl} 1\,0\,0\,1 & \text{Multiplicand} \\ \times\,1\,1\,0\,1 & \text{Multiplier} \\ \hline \left.\begin{array}{r} 1\,0\,0\,1 \\ 0\,0\,0\,0 \\ 1\,0\,0\,1 \\ 1\,0\,0\,1 \end{array}\right\} & \text{Partial Products} \\ \hline 1\,1\,1\,0\,1\,0\,1 & \text{Final product} \end{array}$$

In a digital system, the multiplication operation is performed by repeated additions of all partial products to obtain the final product.

2.5.4 Binary Division

Binary division is obtained using the same procedure as decimal division. An example of binary division is given below:

$$\begin{array}{rl} & 1\,1\,0\,1 \ \text{Quotient} \\ \text{Divisor } 1\,0\,0\,1 & \overline{)\,1\,1\,1\,0\,1\,0\,1} \ \text{Dividend} \\ & 1\,0\,0\,1 \\ & \overline{1\,0\,1\,1} \\ & 1\,0\,0\,1 \\ & \overline{1\,0\,0\,1} \\ & 1\,0\,0\,1 \\ & \overline{} \end{array}$$

2.5.5 Signed Binary Numbers

There are three methods of representing signed binary numbers. They are discussed below.

Sign-Magnitude Representation

In the decimal number system a plus (+) sign is used to denote a positive number and a minus (−) sign for denoting a negative number. The plus sign is usually dropped, and the absence of any sign means that the number has positive value. This representation of numbers is known as ***signed number***. As is well known, digital systems can understand only two symbols, 0 and 1, therefore, we must use the same symbols to indicate the sign of the number also. Normally, an additional bit is used as the ***sign bit*** and it is placed as the most significant bit. ***A 0 is used to represent a positive number and a 1 to represent a negative number.*** For example, an 8-bit signed number $(01000100)_2$ represents a positive number and its value (magnitude) is $(1000100)_2 = (68)_{10}$. The left most 0 (MSB) indicates that the number is positive. On the other hand, in the signed binary form, $(11000100)_2$ represents a negative number with magnitude $(1000100)_2 = (68)_{10}$. The 1 in the left most position (MSB) indicates that the number is negative and the other seven bits give its magnitude. This kind of representation for signed numbers is known as ***signed-magnitude representation***. The user must take care to see the representation used while dealing with the binary numbers.

Table 2.4 gives sign-magnitude numbers represented by 4-bit binary numbers. It can be observed that for an n-bit number, the maximum positive number which can be represented in sign-magnitude representation is $(2^{n-1}-1)$ and the maximum negative number is $-(2^{n-1}-1)$. This method has the disadvantage that zero has dual representation (0000 for +0 and 1000 for −0).

1's Complement Representation

In 1's complement representation, a number is made negative by taking its 1's complement. For example, $(+5)_{10}$ is represented by $(0101)_2$ and $(-5)_{10}$ is represented by $(1010)_2$, where $(1010)_2$ is the 1's complement of (0101)2. In this representation, also MSB is 0 for positive numbers and 1 for negative number. In a positive number, all the bits of the number give its magnitude. In a negative number, 1's complement of the bits of the number give its magnitude.

Table 2.4 Sign-magnitude binary number representation using four bits.

Sign-magnitude binary number	Decimal number
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	−0
1001	−1
1010	−2

1011	−3
1100	−4
1101	−5
1110	−6
1111	−7

Table 2.5 gives 1's complement numbers represented by 4-bit binary numbers. It can be observed that for an n-bit number, the maximum positive number which can be represented in 1's complement representation is $(2^{n-1}-1)$ and the maximum negative number is $-(2^{n-1}-1)$. This method also has the disadvantage that zero has dual representation (0000 for +0 and 1111 for −0).

2's Complement Representation

In 2's complement representation, a number is made negative by taking its 2's complement. For example, $(+5)_{10}$ is represented by $(0101)_2$ and $(-5)_{10}$ is represented by $(1011)_2$, where $(1011)_2$ is the 2's complement of $(0101)_2$. In this representation also, MSB is 0 for positive numbers and 1 for negative numbers. In a positive number, all the bits of the number give its magnitude. In a negative number, 2's complement of the bits of the number give its magnitude.

Table 2.6 gives 2's complement number representation by 4-bit binary numbers. It can be observed that for an n-bit number, the maximum positive number which can be represented in 2's complement representation is $(2^{n-1}-1)$ and the maximum negative number is $-(2^{n-1})$. In 2's complement representation, zero has a unique representation (0000). This method is widely used for representing signed numbers.

Table 2.5 1's complement binary number representation using four bits.

1's complement binary number	Decimal number
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	−7
1001	−6
1010	−5
1011	−4
1100	−3
1101	−2
1110	−1
1111	−0

Table 2.6 2's complement binary number representation using four bits.

2's complement binary number	Decimal number
0000	+0
0001	+1
0010	+2
0011	+3
0100	+4
0101	+5
0110	+6
0111	+7
1000	−8
1001	−7
1010	−6
1011	−5
1100	−4
1101	−3
1110	−2
1111	−1

2.5.6 1's Complement Binary Arithmetic

Binary arithmetic can be easily performed using 1's complement signed binary number representation. But this method is not usually used in digital computers.

Subtraction Using 1's Complement

Binary subtraction can be performed by adding the 1's complement of the subtrahend to the minuend. If an end carry is generated, the answer is positive and add the end carry to the least significant bit (end-around carry) to find the magnitude of the answer. If the end carry is 0, the answer is negative and is in 1's complement form. This method is explained using the examples below:

Example 1: 7 → 0111 Minuend
−5 → (+) 1010 1's complement of subtrahend
+2 (1) 0001
↳(+)1 Add end-around carry
0010

The answer is 0010, which is equivalent to $(+2)_{10}$.

Example 2: 5 → 0101 Minuend
−7 → (+) 1000 1's complement of subtrahend
−2 1101

The end carry is 0. Therefore, the answer is negative and is in 1's complement form. 1's complement of 1101 is 0010. Therefore, the answer is $(-2)_{10}$.

Addition/Subtraction in 1's Complement Representation

The addition/subtraction of signed binary numbers can be performed using 1's complement representation of both the operands. This method is explained using the examples below:

Example 1: 48 – 23 = 48 + (– 23) = + 25
48 → 00110000 8-bit 1's complement representation of + 48
+ (– 23) → (+) 11101000 8-bit 1's complement representation of – 23

+ 25 (1)00011000
→(+) 1 Add end-around carry

00011001 8-bit 1's complement representation of + 25

Example 2: 23 – 48 = 23 + (–48) = – 25

23 → 00010111 8-bit 1's complement representation of + 23
+ (– 48) → (+) 11001111 8-bit 1's complement representation of – 48

– 25 11100110 8-bit 1's complement representation of – 25

Example 3: 48 – (– 23) = 48 + 23 = + 71

48 → 00110000 8-bit 1's complement representation of + 48
+ 23 → (+) 00010111 8-bit 1's complement representation of +23 or –(–23)

+ 71 01000111 8-bit 1's complement representation of + 71

Example 4: – 48 – 23 = (– 48) + (– 23) = – 71
– 48 → 11001111 8-bit 1's complement representation of – 48
– 23 → (+) 11101000 8-bit 1's complement representation of – 23

–71 (1)10110111
→ (+) 1 Add end-around carry

10111000 8-bit 1's complement representation of – 71

From the above examples, we observe the following:

1. If the two operands are of the opposite sign, the result is to be obtained by the rule of subtraction using 1's complement.
2. If the two operands are of the same sign, the sign bit of the result (MSB) is to be compared with the sign bit of the operands. If the sign bits are same, the result is correct and is in 1's complement form. If the sign bits are not same, there is a problem of *overflow*, i.e., the result can not be accommodated using eight bits and the result is to be interpreted suitably. The result in this case will consists of nine bits, i.e., carry and eight bits, and the carry bit will give the sign of the number.

2.5.7 2's Complement Binary Arithmetic

Binary arithmetic can be easily performed using 2's complement signed binary number representation. This method is usually used in digital computers. In this arithmetic, the problem of subtraction is changed to that of addition. Therefore, it is possible to use the circuits designed for binary addition to perform the binary subtraction also. This concept eliminates the need of additional circuits for subtraction, rather the same adder circuit are used for both the operations. This makes design of arithmetic circuits very convenient and cheaper (see Chapter 7).

Subtraction Using 2's Complement

Binary subtraction can be performed by adding the 2's complement of the subtrahend to the minuend. If an end carry is generated, the answer is positive and discard the end carry to find the answer. If the end carry is 0, the answer is negative and is in 2's complement form. This method is explained using the examples below:

Example 1: 7 → 0111 Minuend
– 5 → (+) 1011 2's complement of subtrahend

+ 2 (1)0010
→Discard end carry

The answer is 0010, which is equivalent to $(+2)_{10}$.

Example 2: 5 → 0101 Minuend
– 7 → (+) 1001 2's complement of subtrahend

– 2 1110

The end carry is 0. Therefore, the answer is negative and is in 2's complement form. 2's complement of 1110 is 0010. Therefore, the answer is $(-2)_{10}$.

Addition/Subtraction in 2's Complement Representation

The addition/subtraction of signed binary numbers can be conveniently performed using 2's complement representation of both the operands. This is the method most commonly used when these operations are performed in digital computers. This method is explained using the examples below:

Example 1: 48 – 23 = 48 + (– 23) = + 25

48 → 00110000 8-bit 2's complement representation of + 48
+ (– 23) → (+) 11101001 8-bit 2's complement representation of – 23

+ 25 (1)00011001 8-bit 2's complement representation of + 25
→Discard end carry

Example 2: 23 − 48 = 23 + (−48) = − 25

23 → 00010111 8-bit 2's complement representation of + 23
+ (− 48) → (+) 11010000 8-bit 2's complement representation of − 48

− 25 11100111 8-bit 2's complement representation of − 25

Example 3: 48 − (− 23) = 48 + 23 = + 71

48 → 00110000 8-bit 2's complement representation of + 48
+ 23 → (+) 00010111 8-bit 2's complement representation of + 23 or − (− 23)

+ 71 01000111 8-bit 2's complement representation of + 71

Example 4: − 48 − 23 = (− 48) + (− 23) = − 71

− 48 → 11010000 8-bit 2's complement representation of − 48
− 23 → (+) 11101001 8-bit 2's complement representation of − 23

−71 (1)10111001 8-bit 2's complement representation of − 71
→ Discard end carry

From the above examples, we observe the following:

1. If the two operands are of the opposite sign, the result is to be obtained by the rule of subtraction using 2's complement.
2. If the two operands are of the same sign, the sign bit of the result (MSB) is to be compared with the sign bit of the operands. If the sign bits are same, the result is correct and is in 2's complement form. If the sign bits are not same, there is a problem of *overflow*, i.e., the result can not be accommodated using eight bits and the result is to be interpreted suitably. The result in this case will consists of nine bits, i.e., carry and eight bits, and the carry bit will give the sign of the number.

2.6 Octal Number System

2.6.1 Octal Number Representation

The number system with base (or radix) eight is know as the *octal number system*. In this system, eight digits, 0, 1, 2, 3, 4, 5, 6, and 7 are used to represent numbers. Similar to decimal and binary number systems, it is also a positional number system and every position is assigned a weight of 8^i. For example, $(6327.4051)_8$ is an octal number, which is equivalent to $(3287.5100098)_{10}$.

2.6.2 Octal to Decimal Conversion

An octal number can be converted to decimal number by using the formula

$$(N)_8 = \sum_{i=-m}^{n} d_i 8^i$$

For example,

$$\begin{aligned}(6327.4051)_8 &= 6\times 8^3 + 3\times 8^2 + 2\times 8^1 + 7\times 8^0 + 4\times 8^{-1} \\ &\quad + 0\times 8^{-2} + 5\times 8^{-3} + 1\times 8^{-4} \\ &= 3072 + 192 + 16 + 7 + 4/8 + 0 + 5/512 + 1/4096 \\ &= (3287.5100098)_{10}\end{aligned}$$

2.6.3 Decimal to Octal Conversion

The conversion from decimal to octal is similar to the conversion procedure for decimal to binary conversion. The only difference is that number 8 is used in place of 2 for division in the case of integer part and for multiplication in the case of fractional part. The process is explained below for $(3287.5100098)_{10}$:

Integer part

	Quotient	Remainder
3287/8	410	7 LSD
410/8	51	2
51/8	6	3
6/8	0	6 MSD

Thus $(3287)_{10} = (6327)_8$.

Fractional part

.5100098 × 8 = 4.0800784 → 4 MSD
.0800784 × 8 = 0.6406272 → 0
.6406272 × 8 = 5.1250176 → 5
.1250176 × 8 = 1.0001408 → 1 LSD (up to four places after the radix point)

Thus $(.5100098)_{10} = (.4051)_8$.

Therefore, $(3287.5100098)_{10} = (6327.4051)_8$

From the above example, we observe that the conversion for fractional part may not be exact. In general, an approximate equivalent can be determined by terminating the process of multiplication by eight at the desired places after the radix point.

2.6.4 Octal to Binary Conversion

Octal numbers can be converted into equivalent binary numbers by replacing each octal digit by its 3-bit equivalent binary. Table 2.7 gives octal digits and their binary equivalents.

Table 2.7 Binary and decimal equivalents of octal digits

Octal	Decimal	Binary
0	0	000
1	1	001
2	2	010
3	3	011
4	4	100
5	5	101
6	6	110
7	7	111

The procedure is illustrated for $(736.42)_8$. From Table 2.7, we observe the binary equivalent of 7, 3, 6, 4, and 2 as 111, 011, 110, 100, and 010, respectively. Therefore,

$$(736.42)_8 = (111\ 011\ 110\ .\ 100\ 010)_2 = (111011110.100010)_2$$

2.6.5 Binary to Octal Conversion

Binary numbers can be converted into equivalent octal numbers by making groups of three bits starting from LSB and moving towards MSB for integer part of the number and then replacing each group of three bits by its octal representation. For fractional part, the groupings of three bits are used starting from the binary point. The procedure is illustrated below:

(a) $11001110001.000101111001 = 011\ 001\ 110\ 001\ .\ 000\ 101\ 111\ 001 = (3161.0571)_8$

(b) $1011011110.11001010011 = 001\ 011\ 011\ 110\ .\ 110\ 010\ 100\ 110 = (1336.6246)_8$

(c) $111110001.10011001101 = 111\ 110\ 001\ .\ 100\ 110\ 011\ 010 = (761.4632)_8$

From the above examples we observe that in forming the 3-bit grouping, 0s may be required to complete the first (MSD) group in the integer part and the last (LSD) group in the fractional part.

2.6.6 Decimal to Binary Conversion via Octal and Vice Versa

Large decimal numbers can be easily converted to equivalent binary numbers by converting the decimal number to equivalent octal numbers and then to binary numbers. Similarly, large binary numbers can be converted to equivalent decimal numbers by converting the binary numbers to equivalent octal numbers and then to decimal numbers. The procedure is illustrated using the following examples.

Example 1: $(1325.65625)_{10}$

Integer part

	Quotient	Remainder
1325/8	165	5 LSD
165/8	20	5
20/8	2	4
2/8	0	2 MSD

Thus, $(1325)_{10} = (2455)_8$.

Fractional part

$.65625 \times 8 = 5.25000 \longrightarrow 5$ MSD
$.25000 \times 8 = 2.00000 \longrightarrow 2$ LSD

Thus, $(.65625)_{10} = (52)_8$.

$$\text{Therefore, } (1323.65625)_{10} = (2455.52)_8 = (010\ 100\ 101\ 101\ .\ 101\ 010)_2 = (010100101101.101010)_2$$

Example 2:
$$\begin{aligned}(010100101101.101010)_2 &= (010\ 100\ 101\ 101\ .\ 101\ 010)_2\\ &= (2455.52)_8\\ &= 2\times 8^3 + 4\times 8^2 + 5\times 8^1 + 5\times 8^0 + 5\times 8^{-1} + 2\times 8^{-2}\\ &= 1024 + 256 + 40 + 5 + 5/8 + 2/64\\ &= (1325.65625)_{10}\end{aligned}$$

2.6.7 Octal Arithmetic

Octal arithmetic rules are similar to the decimal or binary arithmetic. Normally, we are not interested in performing octal arithmetic operations using octal representation of numbers. This number system is normally used to enter long strings of binary data into a digital computer. This makes the task of entering binary data into a computer easier. Arithmetic operations can be performed by converting the octal numbers to binary numbers and then using the rules of binary arithmetic. The procedure is illustrated using the following examples.

Example1:
$$\begin{array}{rcr} (23)_8 & \longrightarrow & 010\ 011 \\ +(67)_8 & \longrightarrow & (+)\ 110\ 111 \\ \hline (112)_8 & \longrightarrow & 1\ 001\ 010 \end{array}$$

Example 2:
$$\begin{array}{rcrcrl} (53)_8 & \longrightarrow & 101\ 011 & \longrightarrow & 101\ 011 & \\ -(37)_8 & \longrightarrow & (-)\ 011\ 111 & \longrightarrow & (+)\ 100\ 001 & \text{2's complement of } (37)_8 \\ \hline (14)_8 & & \longrightarrow & & 1\ 001\ 100 & \\ & & & & \hookrightarrow & \text{Discard end carry} \end{array}$$

Example 3:

$$\begin{array}{lllll}
(26)_8 & \longrightarrow & 010\ 110 & \longrightarrow & 010\ 110 \\
-(75)_8 & \longrightarrow & (-)\ 111\ 101 & \longrightarrow & (+)\ 000\ 011 \quad \text{2's complement of } (75)_8 \\
\hline
-(47)_8 & \longrightarrow & & & 011\ 001 \quad \text{2's complement of } (47)_8, \\
 & & & & \quad \text{that is } -(47)_8
\end{array}$$

Multiplication and division can also be performed using the binary representation of octal numbers and then making use of multiplication and division rules of binary numbers.

2.6.8 Application of Octal Number System

In digital systems, binary numbers are required to be entered and certain results or status signals are required to be displayed. It is highly inconvenient to handle long strings of binary numbers. It may cause errors also. Therefore, octal numbers are used for entering the binary data and displaying certain information. Therefore, the knowledge of octal number system is very important for the efficient use of digital systems. For example, the binary number 011111110 can easily be remembered as 376 and can be entered as 376 using octal keys. Since digital circuits can process only 0s and 1s, the octal numbers have to be converted into binary form using special circuit before being processed by the digital circuits.

2.7 Hexadecimal Number System

2.7.1 Hexadecimal Number Representation

Hexadecimal number system is very popular in computer uses. The base for hexadecimal number system is 16. In this system, 16 digits are required to represent the numbers. These digits are numerals 0 through 9 and alphabets A through F. Table 2.8 gives hexadecimal digits with their decimal and binary equivalents. Similar to decimal, binary, and octal number systems, it is also a positional number system and every position is assigned a weight of 16^i. For example, $(3A.2F)_{16}$ is a hexadecimal (hex) number that is equivalent to $(58.1836)_{10}$.

2.7.2 Hexadecimal to Decimal Conversion

A hexadecimal number can be converted to decimal number by using the formula

$$(N)_{16} = \sum_{i=-m}^{n} d_i 16^i$$

For example,

$$\begin{aligned}(3A.2F)_{16} &= 3 \times 16^1 + 10 \times 16^0 + 2 \times 16^{-1} + 15 \times 16^{-2} \\ &= 48 + 10 + 2/16 + 16/156 = (58.1836)_{10}.\end{aligned}$$

The fractional part may not be an exact equivalent and therefore, may give a small error.

Table 2.8 Decimal and binary equivalents of hexadecimal digits

Hexadecimal digits	Decimal numbers	Binary numbers
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

2.7.3 Decimal to Hexadecimal Conversion

For conversion from decimal to hexadecimal, the procedure used in binary and octal systems is applicable, using 16 as the dividing (for integer part) and multiplying (for fractional part) factor. The procedure is illustrated for $(675.6250)_{10}$.

Integer part

	Quotient	Remainder (Hex)
675/16	42	3 LSD
42/16	2	A
2/16	0	2 MSD

Thus, $(675)_{10} = (2A3)_{16}$.

Fractional part

$.625 \times 16 = 10.00 \longrightarrow A$

Thus, $(.625)_{10} = (.A)_{16}$.

Therefore, $(675.625)_{10} = (2A3.A)_{16}$

2.7.4 Hexadecimal to Binary Conversion

Hexadecimal numbers can be converted into equivalent binary numbers by replacing each hexadecimal digit by its equivalent 4-bit binary number. The procedure is illustrated for $(2F9A.5B)_{16}$. From Table 2.8, find the binary equivalent for each hexadecimal digit.

$(2F9A.5B)_{16}$

$= (0010\ 1111\ 1001\ 1010\ .\ 0101\ 1011)_2$

$= (0010111110011010.01011011)_2$

2.7.5 Binary to Hexadecimal Conversion

Binary numbers can be converted into the equivalent hexadecimal numbers by making groups of four bits starting from LSB and moving towards MSB for integer part and then replacing each group of four bits by its hexadecimal representation. For the fractional part, the above procedure is repeated starting from the bit next to the binary point and moving towards the right. The procedure is illustrated for $(10100110101111.00011110101101)_2$.

$(10100110101111.00011110101101)_2$
$= (0010\ 1001\ 1010\ 1111\ .\ 0001\ 1110\ 1011\ 0100)_2$
$= (29AF.1EB4)_{16}$

From the above example, we observe that in forming the 4-bit grouping 0s may be required to complete the first (MSD) group in the integer part and the last (LSD) group in the fractional part.

2.7.6 Hexadecimal to Octal Conversion and Vice Versa

Hexadecimal numbers can be converted to equivalent octal numbers by converting the hexadecimal number to equivalent binary number and then to octal number. Similarly, octal numbers can be converted to equivalent hexadecimal numbers by converting the octal number to equivalent binary number and then to hexadecimal number. The process is illustrated using the following examples.

Example 1: $(A72E.BF85)_{16}$
$= (1010\ 0111\ 0010\ 1110\ .\ 1011\ 1111\ 1000\ 0101)_2$
$= (001\ 010\ 011\ 100\ 101\ 110\ .\ 101\ 111\ 111\ 000\ 010\ 100)_2$
$= (123456.577024)_8$

Example 2: $(247.36)_8$
$= (010\ 100\ 111\ .\ 011\ 110)_2$
$= (0\ 1010\ 0111\ .\ 0111\ 1000)_2$
$= (A7.78)_{10}$

2.7.7 Decimal to Binary Conversion via Hexadecimal and Vice Versa

Large decimal numbers can be easily converted to equivalent binary numbers by converting the decimal number to equivalent hexadecimal number and then to binary number. Similarly, large binary numbers can be converted to equivalent decimal numbers by converting the binary number to equivalent hexadecimal number and then to decimal number. The procedure is illustrated using the following examples.

Example 1: $(1325.65625)_{10}$

Integer part

	Quotient	Remainder (Hex)
1325/16	82	D LSD
82/16	5	2
5/16	0	5 MSD

Thus, $(12325)_{10} = (52D)_{16}$.

Fractional part

$.65625 \times 16 = 10.50000 \rightarrow$ A MSD
$.50000 \times 16 = 8.00000 \rightarrow$ 8 LSD

Thus, $(.65625)_{10} = (.A8)_{16}$.

Therefore, $(1325.65625)_{10} = (52D.A8)_{16}$
$= (0101\ 0010\ 1101\ .\ 1010\ 1000)_2$
$= (010100101101.10101000)_2$

Example 2:
$(010100101101.10101000)_2$
$= (0101\ 0010\ 1101\ .\ 1010\ 1000)_2$
$= (52D.A8)_{16}$
$= 5 \times 16^2 + 2 \times 16^1 + 13 \times 16^0 + 10 \times 16^{-1} + 8 \times 16^{-2}$
$= 1280 + 32 + 13 + 10/16 + 8/256$
$= (1325.65625)_{10}$

2.7.8 Hexadecimal Arithmetic

The rules for arithmetic operations with hexadecimal numbers are similar to the rules for decimal, binary, and octal systems. Normally, we are not interested in performing hexadecimal arithmetic operations using hexadecimal representation of numbers. Similar to octal system, this number system is also used to enter long strings of binary data into a digital computer. Arithmetic operations can be performed by converting the hexadecimal numbers to binary numbers and then using the rule of binary arithmetic. The procedure is illustrated using the following examples.

Example 1:

$$\begin{array}{rcr}
(7F)_{16} & \rightarrow & 0111\ 1111 \\
+\ (BA)_{16} & \rightarrow & (+)\ 1011\ 1010 \\
\hline
(139)_{16} & \rightarrow & 1\ 0011\ 1001
\end{array}$$

Example 2:

$$\begin{array}{rcrcrl}
(3F)_{16} & \rightarrow & 0011\ 1111 & \rightarrow & 0011\ 1111 & \\
-\ (5C)_{16} & \rightarrow & -0101\ 1100 & \rightarrow & +\ 1010\ 0100 & \text{2's complement of } (5C)_{16} \\
\hline
-\ (1D)_{16} & \longrightarrow & & & 1110\ 0011 & \text{2's complement of } (1D)_{16}, \\
 & & & & & \text{that is } -(1D)_{16}
\end{array}$$

Example 3:

$$\begin{array}{rcrcrl}
(C0)_{16} & \rightarrow & 1100\ 0000 & \rightarrow & 1100\ 0000 & \\
-\ (7A)_{16} & \rightarrow & -0111\ 1010 & \rightarrow & +\ 1000\ 0110 & \text{2's complement of } (7A)_{16} \\
\hline
(46)_{16} & \longrightarrow & & & 1\ 0100\ 0110 &
\end{array}$$

$\hookrightarrow$ Discard end carry

Multiplication and division can also be performed using the binary representation of hexadecimal numbers and then making use of multiplication and division rules of binary numbers.

2.7.9 Application of Hexadecimal Number System

Like octal numbers, hexadecimal numbers are also used for entering the binary data and displaying certain information in digital computers. In practice, hexadecimal numbers are more frequently used than octal numbers for this purpose. Therefore, the knowledge of hexadecimal number system is very important for the efficient use of digital systems. For example, the binary number 11111110 can easily be remembered as FE and can be entered as FE using hexadecimal keys.

2.8 Binary Codes

Digital systems represent and manipulate not only binary numbers but also many other discrete elements of information such as numeric, alphabets and special characters. As digital circuits process data in the binary format, these discrete elements of information are represented in digital systems by various binary codes. Although, in every code used the information is represented in binary form, the interpretation of this binary information is possible only if the code in which this information is available is known. For example, the binary number 1000001 represents 65 (decimal), 41 (decimal) in BCD and alphabet A in ACSII code. A user must be very careful about the code being used while interpreting information available in the binary format. Binary codes are also used for error detection and error correction in digital systems.

To represent a group of 2^n distinct elements in a binary code requires a minimum of n bits. This is because it is possible to arrange n bits in 2^n distinct ways. For example, a group of four distinct quantities can be represented by a two-bit code, with each quantity assigned to one of the following bit combinations: 00, 01, 10, 11. A group of eight elements requires a three-bit code, with each element assigned to one and only one of the following: 000, 001, 010, 011, 100, 101, 110, 111. The examples show that the distinct bit combinations of an n-bit code can be found by counting in binary from 0 to $(2^n - 1)$. Some bit combinations are unassigned when the number of elements of the group to be coded is not a multiple of the power of 2. The ten decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 are an example of such a group. A binary code that distinguishes among ten elements must contain at least four bits; three bits can distinguish a maximum of eight elements. Four bits can form 16 distinct combinations, but since only ten digits are coded, the remaining six combinations are unassigned and not used.

Some of the commonly used binary codes are discussed here.

2.8.1 BCD Code

In BCD (binary-coded decimal) code, decimal digits 0 through 9 are represented (coded) by their binary equivalents using four bits as shown in Table 2.9. Each decimal digit of a decimal number is represented by this four-bit code individually. For example, $(23)_{10}$ is represented by 0010 0011 using BCD code, rather than $(10111)_2$. From this it is observed that it requires more number of bits to code a decimal number using BCD code than using the straight binary form. However, in spite of this disadvantage it is very convenient and useful code for input and output operations in digital systems. BCD code is also known as 8421 code; 8, 4, 2, and 1 are the weight of the four bits of the binary code of each decimal digit similar to straight binary number system. Therefore, this is a *weighted* code and arithmetic operations can be performed using this code, which will be discussed in Chapter 7. BCD code is the most commonly encountered decimal code.

2.8.2 Excess-3 Code

In excess-3 code, decimal digit is coded into a four-bit binary code, the code for each decimal digit is obtained by adding decimal 3 to the BCD code of the digit as shown in Table 2.9. For example, decimal digit 2 is coded as 0010 + 0011 = 0101 in Excess-3 code. It is not a weighted code. Like BCD code, each decimal digit of a decimal number is represented by the four-bit Excess-3 code individually. For example, $(23)_{10}$ is represented by 0101 0110 using Excess-3 code. Excess-3 code is a self-complementing code, which means 1's complement of the coded number yields 9's complement of the number itself. For example, Excess-3 code of decimal number 23 is 01010110. Its 1's complement is 10101001, which is Excess-3 code for decimal number 76, which is 9's complement of decimal number 23. The self-complementing property of this code helps considerably in performing subtraction operation in digital systems.

Table 2.9 BCD and Exess-3 codes for decimal digits 0 through 9.

Decimal digit	BCD code $B_3B_2B_1B_0$	Excess-3 code $E_3E_2E_1E_0$
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

2.8.3 Gray Code

Gray code is a very useful code in which a decimal number is represented in binary form in such a way so that each Gray-code number differs from the preceding and the succeeding number by a single bit. For example, the Gray code for decimal number 5 is 0111 and for 6 is 0101. These two codes differ by only one bit position (third from the left). This code is used extensively for shaft encoders because of this property. It is not a weighted code.

The Gray code is a reflected code and can be constructed using the rules given below:

(i) A one-bit Gray code has two code words 0 and 1 representing decimal numbers 0 and 1 respectively.

(ii) (a) An n-bit ($n \geq 2$) Gray code will have first 2^{n-1} Gray codes of $(n-1)$-bits written in order with a leading 0 appended.

(b) The last 2^{n-1} Gray codes will be equal to the Gray code words of an $(n-1)$-bit Gray code, written in reverse order (assuming a mirror placed between first 2^{n-1} and last 2^{n-1} Gray codes) with a leading 1 appended.

The procedure is illustrated below:

1-bit Gray code is constructed using rule (i) as below:

Decimal number	Gray code
0	0
1	1

2-bit Gray code is constructed using rule (ii) and Gray code of 1-bit as below:

Decimal number	Gray code
0	00
1	01
2	11
3	10

3-bit gray code is constructed using 2-bit Gray code as below:

Decimal number	Gray code
0	000
1	001
2	011
3	010
4	110
5	111
6	101
7	100

4-bit gray code is constructed using 3-bit Gray code as below:

Decimal number	Gray code
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

We can also construct the Gray code of a decimal number directly from its binary representation as follows:

Let $b_{n-1}b_{n-2}\cdots b_1b_0$ be the n-bit binary representation of the given decimal number and $g_{n-1}g_{n-2}\cdots g_1g_0$ be the equivalent n-bit Gray code. Then the Gray code can be constructed using the equation

$$g_i = \begin{cases} b_i & \text{if } i = n-1 \\ b_i \oplus b_{i+1} & \text{otherwise} \end{cases}$$

where $\oplus$ indicates modulo-2 sum, according to the following rules

$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

The procedure is illustrated using the following examples.

Example 1: The given decimal number is 10 and its binary equivalent is $b_3b_2b_1b_0$ = 1010. Then

$$g_3 = b_3 = 1$$
$$g_2 = b_2 \oplus b_3 = 0 \oplus 1 = 1$$
$$g_1 = b_1 \oplus b_2 = 1 \oplus 0 = 1$$
$$g_0 = b_0 \oplus b_1 = 0 \oplus 1 = 1$$

and the equivalent Gray code is $g_3g_2g_1g_0$ = 1111.

Example 2: The given decimal number is 14 and its binary equivalent is $b_3b_2b_1b_0$ = 1110. Then

$$g_3 = b_3 = 1$$
$$g_2 = b_2 \oplus b_3 = 1 \oplus 1 = 0$$
$$g_1 = b_1 \oplus b_2 = 1 \oplus 1 = 0$$
$$g_0 = b_0 \oplus b_1 = 0 \oplus 1 = 1$$

and the equivalent Gray code is $g_3g_2g_1g_0$ = 1001.

We can also reconstruct the binary representation of a decimal number from its Gray code:

Let $g_{n-1}g_{n-2}\cdots g_1g_0$ be the given n-bit Gray code and $b_{n-1}b_{n-2}\cdots b_1b_0$ be the n-bit binary representation of its equivalent decimal number. Then the binary representation can be constructed using the equation

$$b_i = \begin{cases} g_i & \text{if } i = n-1 \\ g_i \oplus b_{i+1} & \text{otherwise} \end{cases}$$

The procedure is illustrated using the following examples.

Example 1: The given Gray code is $g_3g_2g_1g_0$ = 1111 and its equivalent binary representation is $b_3b_2b_1b_0$. Then

$$b_3 = g_3 = 1$$
$$b_2 = g_2 \oplus b_3 = 1 \oplus 1 = 0$$
$$b_1 = g_1 \oplus b_2 = 1 \oplus 0 = 1$$
$$b_0 = g_0 \oplus b_1 = 1 \oplus 1 = 0$$

and the binary representation is $b_3b_2b_1b_0$ = 1010, which is equivalent to decimal number 10.

Example 2: The given Gray code is $g_3g_2g_1g_0$ = 1001 and its equivalent binary representation is $b_3b_2b_1b_0$. Then

$$b_3 = g_3 = 1$$
$$b_2 = g_2 \oplus b_3 = 0 \oplus 1 = 1$$
$$b_1 = g_1 \oplus b_2 = 0 \oplus 1 = 1$$
$$b_0 = g_0 \oplus b_1 = 1 \oplus 1 = 0$$

and the binary representation is $b_3b_2b_1b_0$ = 1110, which is equivalent to decimal number 14.

2.8.4 Error-Detection Code

Parity Bit

Binary information can be transmitted from one place to another by any communication medium. Any external noise introduced into the communication medium may change some of the bits from 0 to 1 or vice versa. The purpose of an error-detection code is to detect such bit-reversal errors. One of the most common ways to achieve error detection is by means of a *parity bit*. A parity bit is an extra bit included with a message to make the total number of 1s transmitted either odd (*odd parity*) or even (*even parity*). A message of four bits and a parity bit *P* are shown in Table 2.10. If an odd parity is adopted, the *P* bit is chosen such that the total number of 1s is odd in the five bits that constitute the message and *P*. If an even parity is adopted, the *P* bit is chosen so that the total number of 1s in the five bits is even. In a particular situation, one or the other parity is adopted, with even parity being more common.

Table 2.10 Parity bit.

Odd parity		Even parity	
Message	*P*	Message	*P*
0000	1	0000	0
0001	0	0001	1
0010	0	0010	1
0011	1	0011	0
0100	0	0100	1
0101	1	0101	0
0110	1	0110	0
0111	0	0111	1
1000	0	1000	1
1001	1	1001	0
1010	1	1010	0
1011	0	1011	1
1100	1	1100	0
1101	0	1101	1
1110	0	1110	1
1111	1	1111	0

The parity bit is helpful in detecting errors during the transmission of information from one place to another. This is done in the following manner. An even parity is generated in the sending end for each message transmission. The message, together with the parity bit, is transmitted to its destination. The parity of the received data is checked in the receiving end. If the parity of the received information is not even, it means that at least one bit has changed value during the transmission. This method detects one, three, or any odd combination of errors in each message that is transmitted. An even combination of errors is undetected. Additional error-detection schemes may be needed to take care of an even combination of errors. If an error is detected, the data is retransmitted.

Cyclic Redundancy Check (CRC)

One of the most common and popular error-detecting codes is the *Cyclic Redundancy Check (CRC)*. In this code, an *n*-bit sequence *F*, called *Frame Check Sequence (FCS)*, is appended with a *k*-bit message *M* to form the (*k* + *n*)-bit transmitted message *T*. The FCS *F* is determined in such a way that the transmitted message *T* is exactly divisible by an (*n* + 1)-bit pattern *P*. The receiver then divides the received message by the pattern *P* and if there is no remainder, the received message is assumed to be error-free.

In CRC process all arithmetic is done as modulo-2 arithmetic. Modulo-2 arithmetic uses binary arithmetic with no carries, which is shown in Table 2.11. The modulo-2 arithmetic is illustrated using the following example.

Example: The modulo-2 addition, subtraction, and multiplications are showm below:

$$\begin{array}{r} 1111 \\ +1010 \\ \hline 0101 \end{array} \qquad \begin{array}{r} 1110 \\ -0101 \\ \hline 1011 \end{array} \qquad \begin{array}{r} 11001 \\ \times\ 11 \\ \hline 11001 \\ 11001 \\ \hline 101011 \end{array}$$

Table 2.11 Modulo-2 arithmetic.

a b	a + b	a − b	a × b
0 0	0	0	0
0 1	1	1	1
1 0	1	1	1
1 1	0	0	1

Now, we see how the *n*-bit FCS is determined from *k*-bit message *M* and (*n* + 1)-bit pattern *P* to form the (*k* + *n*)-bit transmitted message *T*. The (*k* + *n*)-bit transmitted message *T* is formed by concatenating *k*-bit message *M* and *n*-bit FCS *F*.

T can be formed by shifting *M* *n*-bits left with padded out the vacant positions by 0 and then adding *F* with the shifted *M*. This operation can be mathematically expressed as

$$T = 2^n M + F$$

This is true, because shifting a number *n*-bit left is equivalent to multiplying the number by 2^n. The CRC process requires that *T/P* has no remainder. Suppose that we divided $2^n M$ by *P*

$$\frac{2^n M}{P} = Q + \frac{R}{P}$$

There is a quotient *Q* and a remainder *R*. Because division is modulo-2, the remainder *R* is always at least one bit shorter than the divisor *P*. That means, if *P* is an (*n* + 1)-bit pattern, the *R* will be a number of at most *n*-bit. We will use this remainder *R* as our FCS *F*. Thus

$$T = 2^n M + R$$

Now divide *T* by *P*:

$$\frac{T}{P} = \frac{2^n M + R}{P} = \frac{2^n M}{P} + \frac{R}{P} = Q + \frac{R}{P} + \frac{R}{P} = Q$$

This is true, as any binary number added to itself modulo-2 yields zero. There is no remainder, and, therefore, *T* is exactly divisible by *P*. Thus FCS is easily generated by dividing $2^n M$ by *P* and using the remainder as the FCS. On reception the transmitted message *T* is divided by *P* and if there is no remainder, then the message is error-free, otherwise the transmitted message has error. The CRC process is illustrated using the following example.

Example: Message *M* = 1010001101 (10 bits) and Pattern *P* = 110101 (6 bits). The FCS will be a 5-bit number.

$2^5 M = 101000110100000$. Dividing $2^5 M$ by P we get

```
                             1 1 0 1 0 1 0 1 1 0
P ──► 1 1 0 1 0 1 │1 0 1 0 0 0 1 1 0 1 0 0 0 0 0 ◄── 2^n M
                  │1 1 0 1 0 1
                   -----------
                     1 1 1 0 1 1
                     1 1 0 1 0 1
                     -----------
                         1 1 1 0 1 0
                         1 1 0 1 0 1
                         -----------
                             1 1 1 1 1 0
                             1 1 0 1 0 1
                             -----------
                                 1 0 1 1 0 0
                                 1 1 0 1 0 1
                                 -----------
                                   1 1 0 0 1 0
                                   1 1 0 1 0 1
                                   -----------
                                       0 1 1 1 0 ◄── R
```

The remainder is added to $2^n M$ to give

```
2^n M  ──►  1 0 1 0 0 0 1 1 0 1 0 0 0 0 0
R      ──►                      + 0 1 1 1 0
-------------------------------------------
T      ──►  1 0 1 0 0 0 1 1 0 1 0 1 1 1 0
```

which is transmitted. If there is no error, the receiver receives T intact. The received message is divided by P:

```
                             1 1 0 1 0 1 0 1 1 0 ◄── Q
P ──► 1 1 0 1 0 1 │1 0 1 0 0 0 1 1 0 1 0 1 1 1 0 ◄── T
                  │1 1 0 1 0 1
                   -----------
                     1 1 1 0 1 1
                     1 1 0 1 0 1
                     -----------
                         1 1 1 0 1 0
                         1 1 0 1 0 1
                         -----------
                             1 1 1 1 1 0
                             1 1 0 1 0 1
                             -----------
                                 1 0 1 1 1 1
                                 1 1 0 1 0 1
                                 -----------
                                   1 1 0 1 0 1
                                   1 1 0 1 0 1
                                   -----------
                                               0 ◄── R
```

Because there is no remainder, it is assumed that there have been no error.

The pattern P is chosen to be one bit larger than the desired FCS, and the exact bit pattern chosen depends on the type of errors expected. At minimum, both the MSB and LSB of P must be 1. Four versions of P are widely used:

CRC-12: 1100000001111
CRC-16: 11000000000000101
CRC-CCIT: 10001000000100001
CRC-32: 100000100110000010001110110110111

20.8.5 ASCII Character Code

Many applications of digital systems require the handling of data not only of numbers, but also of letters. For instance, an insurance company with thousands of policyholders will use a computer to process its files. To represent the names and other pertinent information, it is necessary to formulate a binary code for the letters of the alphabet. In addition, the same binary code must represent numerals and special characters such as \$. An alphanumeric character set is a set of elements that include the 10 decimal digits, the 26 letters of the alphabet, and a number of special characters.

The standard binary code for the alphanumeric characters is ASCII (American Standard Code for Information Interchange). It uses seven bits to code 128 characters, as shown in Table 2.12. The seven bits of the code are designated by b_0 through b_6, with b_6 being the MSB. The letter A, for example, is represented in ASCII as 1000001 (column 100, row 0001). The ASCII code contains 94 graphic characters that can be printed and 34 nonprinting characters used for various control functions. The graphic characters consists of the 26 upper case letters (A through Z), the 26 lower case letters (a through z), the 10 numerals (0 through 9), and 32 special printable characters such as %, *, and \$.

Table 2.12 American Standard Code for Information Interchange.

	$b_6b_5b_4$							
$b_3b_2b_1b_0$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	-	o	DEL

ASCII is a seven-bit code, but most computers manipulate an eight-bit quantity as a single unit called *byte*. Therefore, ASCII characters most often are stored one per byte. The extra bit is sometimes used for other purposes, depending on the application. For example, some printers recognize eight-bit ASCII characters with the MSB set to 0. Additional 128 eight-bit characters with the MSB set to 1 are used for other symbols such as the Greek alphabet or italic type font. When used in data communication, the MSB may be employed to indicate the parity of the character.

Exercise

2.1 List the first 16 numbers in base 12. Use the letters A and B to represent the last two digits.

2.2 What is the largest binary number that can be obtained with 16 bits? What is its decimal equivalent?

2.3 Convert the following binary numbers to decimal equivalent.

(a) 10101 (b) 1011.101 (c)101101.1101 (d)111001
(e) 101001 (f) 11111110 (g) 1100100 (h) 1101.0011
(i) 1010.1010 (j) 0.111000 (k) 101110 (l) 1110101.11
(m) 110110100

2.4 Convert the following decimal numbers to binary, octal, and hexadecimal equivalents.

(a) 367 (b) 235.8125 (c) 10.0101 (d) 37
(e) 255 (f) 15 (g) 26.25 (h) 11.75 (i) 0.1
(j) 1231 (k) 673.23 (l) 10^4 (m) 1998
(n) 7562.45 (o) 1938.257 (p) 175.175

2.5 Convert the following numbers from decimal to octal and then to binary.

(a) 375 (b) 249 (c) 27.125

2.6 Convert the following binary numbers to octal and then to decimal.

(a) 11011100.101010 (b) 01010011.010101
(c) 10110011 (d) 11010111

2.7 Convert the following numbers from decimal to hexadecimal and then to binary.

(a) 375 (b) 249 (c) 27.125

2.8 Convert the following binary numbers to hexadecimal and then to decimal.

(a) 11011100.101010 (b) 01010011.010101 (c) 10110011

2.9 Convert the following binary numbers to octal and hexadecimal numbers.

(a) 1010111 (b) 111101000.0111 (c) 11101

2.10 Convert the following octal numbers to decimal, binary, and hexadecimal.

(a) 623 (b) 443

2.11 Convert the following hexadecimal numbers to decimal, binary, and octal.

(a) F3A7C2 (b) 2AC5

2.12 Convert the following numbers with the indicated bases to decimal.

(a) $(12121)_3$ (b) $(4310)_5$ (c) $(50)_7$ (d) $(198)_{12}$

2.13 The $(r-1)$'s complement of base-6 numbers is called the 5's complement.

(a) Determine a procedure for obtaining the 5's complement of base-6 numbers.

(b) Obtain the 5's complement of $(543210)_6$.

2.14 Perform the following binary arithmetic operations.

(a) 1011.01 + 0101.10 + 1000.01 (b) 11011.101 − 1101.011
(c) 11.01 × 10.11 (d) 1011 + 1101
(e) 1010.1101 + 101.01 (f) 110110 × 110101
(g) 11111111 ÷ 101

2.15 Perform the following arithmetic operations in octal number system.

(a) 237 + 542 (b) 145 × 24 (c) 426 − 247

2.16 Perform the following arithmetic operations in hexadecimal number system.

(a) 23A + 5C2 (b) 14B × 24 (c) 42A − 2F7

2.17 Find the 9's complement of the following 8-digit decimal numbers.

(a) 12349876 (b) 00980100 (c) 90009951 (d) 00000000

2.18 Find the 10's complement of the following 6-digit decimal numbers.

(a) 123900 (b) 090657 (c) 100000 (d) 000000

2.19 Write the 10's and 2's complement of the following decimal numbers.

(a) 3578 (b) 9090 (c) 136.8

2.20 Write the 2's and 1's complement of the following binary numbers.

(a) 1101.01 (b) 1010101 (c) 00001
(d) 10101110 (e) 10000001 (f) 10000000
(g) 00000001 (h) 00000000

2.21 Perform subtraction with the following unsigned decimal numbers by taking both the 10's and 9's complement of the subtrahend.

(a) 5250 − 1321 (b) 1753 − 8640
(c) 20 − 100 (d) 1200 − 250

2.22 Perform the following subtractions using ordinary, 1's complement, and 2's complement methods.

(a) 1010 − 01101 (b) 0010 − 1010
(c) 01000 − 01001 (d) 01100 − 00011
(e) 0011.1001 − 0001.1110 (f) 11010 − 10000
(g) 11010 − 1101 (h) 100 − 110000
(i) 1010100 − 1010100

2.23 The binary numbers listed have a sign in the leftmost position and, if negative, are in 2's complement form. Perform the arithmetic operations indicated.

(a) 101011 + 111000 (b) 001110 + 110010 (c) 111001 − 001010 (d) 101011 − 100110

2.24 Perform the following arithmetic operations in binary using both the signed-1's-complement representation and signed-2's-complement representation for negative numbers.

(a) (+ 42) + (− 13) (b) (− 42) − (− 13)

2.25 Determine the value of base x if $(211)_x = (152)_8$.

2.26 Noting that $3^2 = 9$, formulate a simple procedure for converting base-3 numbers directly to base-9. Use the procedure to convert $(2110201102220112)_3$ to base-9.

2.27 Write the following decimal numbers in BCD and Excess-3 codes.

(a) 4793 (b) 46 (c) 327.89 (d) 20.305
(e) 13597 (f) 93286 (g) 99880 (h) 8620
(i) 2421

2.28 Encode the following decimal numbers in Gray code.

(a) 46 (b) 30 (c) 55 (21

2.29 Decode the following Gray codes to decimal numbers.

(a) 1010111 (b) 0011001 (c) 1111111 (d) 0000000 (e) 0000001

2.30 If the Gray code for a word is $g_{n-1}g_{n-2}\cdots g_i\cdots g_1g_0$, prove that the ith bit of the corresponding binary number $b_{n-1}b_{n-2}\cdots b_i\cdots b_1b_0$ is given by

$$b_i = \begin{cases} g_i & \text{if } i = n-1 \\ g_i \oplus g_{i+1} \oplus \cdots \oplus g_{n-2} \oplus g_{n-1} & \textit{otherwise} \end{cases}.$$

2.31 List the ten BCD digits with both odd and even parity in the leftmost position.

2.32 Attach an even parity bit as MSB for ASCII code.

2.33 Attach an odd parity bit as MSB for ASCII code.

2.34 Encode the following texts in ASCII using 8-bit code with the leftmost bit always 0.

(a) Dhanmondi, Dhaka - 1205.

(b) 4 + 5 − 2

(c) Digital Logic Design

2.35 Write your full name in ASCII using 8-bit code with the leftmost bit always 0. Include a space between names and a period after a middle initial.

2.36 Decode the following ASCII code:

1001010 1101111 1101000 1101110 0100000 1000100 1101111 1100101

2.37 Show the bit configuration that represents the decimal number 295 (a) in binary, (b) in BCD, (c) in Excess-3, and (d) in ASCII

2.38 How many printing characters are there in ASCII? How many of them are not letters or numerals?

2.39 The state of a 12-bit register is 010110010111. What is its content if it represents: (a) three decimal digits in BCD; (b) three decimal digits in the Excess-3 code?

2.40 Find the number of bits required to encode:

(a) 56 elements of information and (b) 130 elements of information

2.41 For $P = 110011$ and $M = 11100011$, find the CRC code.

2.42 For $P = 11001$ and $M = 10011011100$, find the CRC code.

2.43 How many bits of memory are required for storing 100 names of a group of people, assuming that no name occupies more than 20 characters (including spaces)? Assume 7-bit ASCII code with parity bit.

2.44 A line printer is capable of printing 132 characters in a single line and each character is represented by ASCII code. How many bits are required to print each line?

Further Reading

1. Zvi Kohavi, *Switching and Finite Automata Theory*, 2nd ed., McGraw-Hill Higher Education, 1990.
2. Richard S. Sandige, *Modern Digital Design*, McGraw-Hill Publishing Company, 1990.
3. R. P. Jain, *Modern Digital Electronics*, 2nd ed., Tata McGraw-Hill Publishing Company Limited, India, 1998.
4. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
5. D. C. Green, *Digital Electronics*, 5th ed., Pearson Education, 2001.
6. Ronald J. Tocci and Neal S. Widmer, *Digital Systems: Principles and Applications*, 8th ed., Prentice-Hall, Inc., 2001.
7. V. K. Jain, *Switching Theory and Digital Electronics*, Khanna Publishers, India, 2002.
8. M. Morris Mano, *Digital Design*, 3rd ed., Pearson Education, 2002.

CHAPTER 3
BOOLEAN ALGEBRA

3.1 Introduction

As pointed out in Chapter 1, digital logic circuits operate in the binary mode, where each input and output is either 0 or 1. This characteristic of logic circuits allows us to use *Boolean algebra* as a tool for analysis and design of digital systems. In this chapter we shall discuss Boolean algebra in details.

3.2 Axiomatic Definition of Boolean Algebra

Boolean algebra, like any other deductive mathematical system, may be defined with a set of elements, a set of operators, and a number of unproved axioms or postulates. A *set* of elements is any collection of objects having a common property. If S is a set, and x and y are certain objects, then $x\in S$ denotes that x is a member of the set S, and $y\notin S$ denotes that y is not an element of S. A set with a denumerable number of elements is specified by braces: $A=\{1,2,3,4\}$, i.e., the elements of set A are the numbers 1, 2, 3, and 4. A *binary operator* defined on a set S of elements is a rule that assigns to each pair of elements from S a unique element from S. As an example, consider the relation $a*b=c$. We say that * is a binary operator if it specifies a rule for finding c from the pair (a,b) and also if $a,b,c\in S$. However, * is not a binary operator if $a,b\in S$, while the rule finds $c\notin S$. The *postulates* of a mathematical system form the basic assumptions from which it is possible to deduce the rules, theorems, and properties of the system.

In 1854 George Boole introduced a systematic treatment of logic and developed for this purpose an algebraic system now called *Boolean algebra*. In 1938 C. E. Shannon introduced a two-valued Boolean algebra called *switching algebra*, in which he demonstrated that the properties of bistable electrical switching circuits could be represented by this algebra. For the formal definition of Boolean algebra, we shall employ the postulates formulated by E. V. Huntington in 1904.

3.2.1 Postulates of Boolean Algebra

Boolean algebra is an algebraic structure defined on a set of elements B together with two binary operators + and · provided the following (Huntington) postulates are satisfied:

1. (a) Closure with respect to the operator +: For $x, y\in B, x+y=c\in B$.

   (b) Closure with respect to the operator ·: For $x, y\in B, x\cdot y=c\in B$.

2. (a) An identity element with respect to +, designated by 0: For $x\in B, x+0=0+x=x$.

   (b) An identity element with respect to ·, designated by 1: For $x\in B, x\cdot 1=1\cdot x=x$.

3. (a) Commutative with respect to +: For $x, y\in B, x+y=y+x$.

   (b) Commutative with respect to ·: For $x, y\in B, x\cdot y=y\cdot x$.

4. (a) · is distributive over +: For $x, y, z\in B, x\cdot(y+z)=(x\cdot y)+(x\cdot z)$.

   (b) + is distributive over ·: For $x, y, z\in B, x+(y\cdot z)=(x+y)\cdot(x+z)$.

5. For every element $x\in B$, there exists an element $x'\in B$ (called the *complement* of x) such that:

   (a) $x+x'=1$, and

   (b) $x\cdot x'=0$.

6. There exists at least two elements $x, y\in B$ such that $x\neq y$.

It is important to distinguish between the elements of the set of an algebraic structure and the variables of an algebraic system. In Boolean algebra, one defines the elements of the set B, and variables such as x, y, z are merely symbols that represent the elements. At this point, it is important to realize that in order to have a Boolean algebra, one must show:

1. the elements of the set B,
2. the rules of operation for the two binary operators, and
3. that the set of elements B, together with the two operators, satisfies the six Huntington postulates.

One can formulate many Boolean algebra, depending on the choice of elements of B and the rules of operation. In this book, we deal only with a two-valued Boolean algebra, i.e., one with only two elements.

3.2.2 Tow-valued Boolean Algebra

A two-valued Boolean algebra is defined on a set of two elements, $B=\{0,1\}$, with rules for the two binary operators + and · as shown in Table 3.1 (the rule for the complement operator is for verification of postulate 5):

Table 3.1 Two-valued Boolean operator tables.

$x\ y$	$x\cdot y$
0 0	0
0 1	0
1 0	0
1 1	1

$x\ y$	$x+y$
0 0	0
0 1	1
1 0	1
1 1	1

x	x'
0	1
1	0

These rules are exactly the same as the AND, OR, and NOT operations, respectively, defined in Table 1.1. We must now show that the Huntington postulates are valid for the set $B=\{0,1\}$ and the two binary operators defined in Table 3.1.

1. *Closure* is obvious from the two binary operators defined in Table 3.1, since the result of each operation is either 0 or 1 and $0,1\in B$.

2. From the two binary operators of Table 3.1, we see that

   (a) $0+0=0$ and $0+1=1+0=1$
   (b) $1\cdot 1=1$ and $1\cdot 0=0\cdot 1=0$

   which establishes the two *identity elements* 0 for + and 1 for · as defined by postulate 2.
3. The *commutative* laws are obvious from the symmetry of the binary operators in Table 3.1.
4. (a) The *distributive law* $x\cdot(y+z)=(x\cdot y)+(x\cdot z)$ can be shown to hold true from the binary operators of Table 3.1 by forming a truth table of all possible values of x, y, and z. For each combination, we derive $x\cdot(y+z)$ and show that the value is the same as $(x\cdot y)+(x\cdot z)$, as shown in Table 3.2.

Table 3.2 Proof of the distributive law $x\cdot(y+z)=(x\cdot y)+(x\cdot z)$

$x\,y\,z$	$y+z$	$x\cdot(y+z)$	$x\cdot y$	$x\cdot z$	$(x\cdot y)+(x\cdot z)$
000	0	0	0	0	0
001	1	0	0	0	0
010	1	0	0	0	0
011	1	0	0	0	0
100	0	0	0	0	0
101	1	1	0	1	1
110	1	1	1	0	1
111	1	1	1	1	1

(b) The *distributive law* $x+(y\cdot z)=(x+y)\cdot(x+z)$ can be shown to hold true from the binary operators of Table 3.1 by forming a truth table of all possible values of x, y, and z. For each combination, we derive $x+(y\cdot z)$ and show that the value is the same as $(x+y)\cdot(x+z)$, as shown in Table 3.3.

Table 3.3 Proof of the distributive law $x+(y\cdot z)=(x+y)\cdot(x+z)$

$x\,y\,z$	$y\cdot z$	$x+(y\cdot z)$	$x+y$	$x+z$	$(x+y)\cdot(x+z)$
000	0	0	0	0	0
001	0	0	0	1	0
010	0	0	1	0	0
011	1	1	1	1	1
100	0	1	1	1	1
101	0	1	1	1	1
110	0	1	1	1	1
111	1	1	1	1	1

5. From the complement operator in Table 3.1, it is easily shown that:

   (a) $x+x'=1$, since $0+0'=0+1=1$ and $1+1'=1+0=1$.
   (b) $x\cdot x'=0$, since $0\cdot 0'=0\cdot 1=0$ and $1\cdot 1'=1\cdot 0=0$.

   which verifies postulate 5.
6. Postulate 6 is satisfied because the two-valued Boolean algebra has two distinct elements 0 and 1 with $0\neq 1$.

We have just established a two-valued Boolean algebra having a set of two elements, 0 and 1, two binary operators with operation rules equivalent to the AND and OR operations, and a complement operation equivalent to the NOT operation. From here on, we shall drop the adjective "two-valued" from Boolean algebra in subsequent discussion.

3.3 Basic Theorems and Properties of Boolean Algebra

3.3.1 Duality Principle

The Huntington postulates have been listed in pairs and designated by part (a) and part (b). One part may be obtained from the other if the binary operators and the identity elements are interchanged. This important property of Boolean algebra is called *duality principle*. It states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged. In a two-valued Boolean algebra, the identity elements and the elements of the set *B* are the same, 0 and 1. The duality principle has many applications. If the *dual* of an algebraic expression is desired, we simply interchange OR and AND operators and replace 0s by 1s and 1s by 0s.

3.3.2 Basic Theorems of Boolean Algebra

Table 3.4 lists six theorems of Boolean algebra and four of its postulates. The notation is simplified by omitting the · whenever this does not lead to confusion. The theorems and postulates listed are the most basic relationships in Boolean algebra. The theorems, like the postulates, are listed in pairs; each relation is the dual of the one paired with it. The postulates are basic axioms of the algebraic structure and need no proof. The theorems must be proved from the postulates.

Table 3.4 Postulates and theorems of Boolean algebra.

	(a)	(b)
Postulate 2	$x+0=x$	$x\cdot 1=x$
Postulate 3 (*commutative law*)	$x+y=y+x$	$xy=yx$
Postulate 4 (*distributive law*)	$x(y+z)=xy+xz$	$x+yz=(x+y)(x+z)$
Postulate 5	$x+x'=1$	$x\cdot x'=0$
Theorem 1	$x+x=x$	$x\cdot x=x$
Theorem 2	$x+1=1$	$x\cdot 0=0$
Theorem 3 (*involution law*) $(x')'=x$		
Theorem 4 (*associative law*)	$x+(y+z)=(x+y)+z$	$x(yz)=(xy)z$
Theorem 5 (*DeMorgan's theorem*)	$(x+y)'=x'y'$	$(xy)'=x'+y'$
Theorem 6 (*absorption law*)	$x+xy=x$	$x(x+y)=x$; $x+\bar{x}\cdot y=x+y$

The proofs of the theorems are presented below. At the right is listed the number of the postulates and previously proved theorems which justifies each step of the proof.

Theorem 1(a): $x+x=x$.

$x+x=(x+x)\cdot 1$	by postulate:	2(b)
$=(x+x)(x+x')$		5(a)
$=x+xx'$		4(b)
$=x+0$		5(b)
$=x$		2(a)

Theorem 1(b): $x\cdot x=x$.

$x\cdot x=xx+0$	by postulate:	2(a)
$=xx+xx'$		5(b)
$=x(x+x')$		4(a)
$=x\cdot 1$		5(a)
$=x$		2(b)

Note that theorem 1(b) is the dual of theorem 1(a) and that each step of the proof in part (b) is the dual of part (a). Any dual theorem can be similarly derived from the proof of its corresponding pair.

$$\begin{aligned} x+\bar{x}\cdot y &= x(1+y)+\bar{x}\cdot y \\ &= x+x\cdot y+\bar{x}\cdot y \\ &= x+y(x+\bar{x}) = x+y \end{aligned}$$

Theorem 2(a): $x+1=1$.

$x+1=(x+1)\cdot 1$	by postulate:	2(b)
$=1\cdot(x+1)$		3(b)
$=(x+x')(x+1)$		5(a)
$=x+x'\cdot 1$		4(b)
$=x+x'$		2(b)
$=1$		5(a)

Theorem 2(b): $x\cdot 0=0$ by duality.

Theorem 3: $(x')'=x$.

From postulate 5, we have $x+x'=1$ and $x\cdot x'=0$, which defines the complement of x as x'. Therefore, the complement of x' is x, that is, $(x')'$ is x.

Theorem 4(a): $x+(y+z)=(x+y)+z$.

$x+(y+z)=x+(y+z)\cdot 1$	by postulate:	2(b)
$=x+1\cdot(y+z)$		3(b)
$=x+1\cdot y+1\cdot z$		4(a)
$=x+y\cdot 1+z\cdot 1$		3(b)
$=x\cdot 1+y\cdot 1+z$		2(b)
$=1\cdot x+1\cdot y+z$		3(b)
$=1\cdot(x+y)+z$		4(a)
$=(x+y)\cdot 1+z$		3(b)
$=(x+y)+z$		2(b)

Theorem 4(b): $x(yz)=(xy)z$ by duality.

Theorem 5(a): $(x+y)'=x'y'$.

We are to prove that the complement of $(x+y)$ is $x'y'$. From postulate 5, we have to prove $(x+y)+x'y'=1$ and $(x+y)\cdot x'y'=0$.

$(x+y)+x'y'=x+(y+x'y')$	by theorem	4(a)
$=x+(y+x')(y+y')$	by postulate	4(b)
$=x+(y+x')\cdot 1$	by postulate	5(a)

$$\begin{aligned}
&= x+1\cdot(y+x') && \text{by postulate} && 3(b)\\
&= x+1\cdot y+1\cdot x' && \text{by postulate} && 4(a)\\
&= x+y\cdot 1+x'\cdot 1 && \text{by postulate} && 3(b)\\
&= x+y+x' && \text{by postulate} && 2(b)\\
&= y+x+x' && \text{by postulate} && 3(a)\\
&= y+1 && \text{by postulate} && 5(a)\\
&= 1 && \text{by theorem} && 2(a)
\end{aligned}$$

$$\begin{aligned}
(x+y)\cdot x'y' &= x'y'\cdot(x+y) && \text{by postulate} && 3(b)\\
&= x'y'x+x'y'y && \text{by postulate} && 4(a)\\
&= y'x'x+x'yy' && \text{by postulate} && 3(b)\\
&= y'xx'+x'yy' && \text{by postulate} && 3(b)\\
&= y'\cdot 0+x'\cdot 0 && \text{by postulate} && 5(b)\\
&= 0+0 && \text{by theorem} && 2(b)\\
&= 0 && \text{by theorem} && 1(a)
\end{aligned}$$

Theorem 5(b): $(xy)' = x' + y'$ by duality.

Theorem 6(a): $x + xy = x$.

$$\begin{aligned}
x+xy &= x\cdot 1+xy && \text{by postulate} && 2(b)\\
&= x(1+y) && \text{by postulate} && 4(a)\\
&= x(y+1) && \text{by postulate} && 3(a)\\
&= x\cdot 1 && \text{by theorem} && 2(a)\\
&= x && \text{by postulate} && 2(b)
\end{aligned}$$

Theorem 6(b): $x(x+y) = x$ by duality.

DeMorgan's Theorem for More Than Two Variables

DeMorgan's theorem can be extended to three or more variables. The three-variable form of the first DeMorgan's theorem is derived below.

$$\begin{aligned}
(A+B+C)' &= (A+X)' && \text{let } B+C=X\\
&= A'X' && \text{by theorem 5(a)}\\
&= A'\cdot(B+C)' && \text{substituting } B+C=X\\
&= A'\cdot(B'C') && \text{by theorem 5(a)}\\
&= A'B'C' && \text{by theorem 4(b)}
\end{aligned}$$

$(ABC)' = A' + B' + C'$ by duality.

DeMorgan's theorem for any number of variables resemble in form the two-variable case and can be derived by successive substitution similar to the method used in the above derivation. These theorems can be generalized as follows:

$$(A+B+C+D+E+\cdots)' = A'B'C'D'E'\cdots$$
$$(ABCDE\cdots)' = A'+B'+C'+D'+E'+\cdots$$

3.3.3 Operator Precedence

The operator precedence for evaluating Boolean expression is (1) parentheses, (2) NOT, (3) AND, and (4) OR. In other words, the expression inside the parentheses must be evaluated before all other operations. The next operation that holds precedence is the NOT, then follows the AND, and finally the OR. For example, the evaluation of the Boolean expression $xy' + z(x+y)$ for $x = 0$, $y = 1$, and $z = 0$ is shown below:

$$\begin{aligned}
xy'+z(x+y) &= 0\cdot 1'+0\cdot(0+1)\\
&= 0\cdot 1'+0\cdot 1 && \text{evaluation of parenthesis}\\
&= 0\cdot 0+0\cdot 1 && \text{evaluation of NOT}\\
&= 0+0 && \text{evaluation of AND}\\
&= 0 && \text{evaluation of OR}
\end{aligned}$$

3.4 Boolean Function

3.4.1 Algebraic Expression of Boolean Function

A binary variable can take the value of either 0 or 1. A *Boolean function* is an expression formed with binary variables, the two binary operators OR and AND, the unary operator NOT, parentheses, and equal sign. For a given value of the variables, the functional value of the expression can be either 0 or 1. The following is an algebraic expression of a Boolean function:

$$F_1 = xyz'$$

The function F_1 is equal to 1 if $x=1$ and $y=1$ and $z=0$; otherwise $F_1 = 0$.

3.4.2 Truth Table Representation of Boolean Function

A Boolean function may be represented in a truth table. To represent a Boolean function in a truth table, we need a list of the 2^n combinations of 0 and 1 of the n binary variables, and a column showing the functional value of either 0 or 1 for each combination of values of the n variables. As shown in Table 3.5, there are eight possible distinct combinations for assigning bits of three variables. The column labeled F_1 contains either a 0 or a 1 for each of these combinations. The table shows that the function F_1 is equal to 1 only when $x=1$, $y=1$, and $z=0$. It is equal to 0 otherwise. Consider now the function:

$$F_2 = x + y'z$$

The function $F_2 = 1$ if $x = 1$; or $y = 0$ and $z = 1$. In Table 3.5, $x = 1$ in the last four rows; $y = 0$ and $z = 1$ in rows 001 and 101. The latter combination applies also for $x = 1$. Therefore, there are five combinations that make $F_2 = 1$.

As a third example, consider the function:

$$F_3 = x'y'z + x'yz + xy'$$

The function $F_3 = 1$ if $x = 0$ and $y = 0$ and $z = 1$; or $x = 0$ and $y = 1$ and $z = 1$; or $x = 1$ and $y = 0$. $x = 1$ and $y = 0$ in rows 100 and 101. This is shown in Table 3.5 with four 1s and four 0s.

As a fourth example, consider the function:

$$F_4 = xy' + x'z$$

The function $F_4 = 1$ if $x = 1$ and $y = 0$; or $x = 0$ and $z = 1$. $x = 1$ and $y = 0$ in rows 100 and 101. $x = 0$ and $z = 1$ in rows 001 and 011. This is shown in Table 3.5 with four 1s and four 0s.

Table 3.5 Truth tables for $F_1 = xyz'$, $F_2 = x + y'z$, $F_3 = x'y'z + x'yz + xy'$, and $F_4 = xy' + x'z$

$x\ y\ z$	F_1	F_2	F_3	F_4
000	0	0	0	0
001	0	1	1	1
010	0	0	0	0
011	0	0	1	1
100	0	1	1	1
101	0	1	1	1
110	1	1	0	0
111	0	1	0	0

3.4.3 Non-unique Algebraic Expression of Boolean Function

From Table 3.5, we find that F_3 is equal to F_4, since both have identical 0s and 1s for each combination of values of the three binary variables. In general, two functions of n binary variables are said to be equal if they have the same value for all possible 2^n combinations of the n variables. Though F_3 is equal to F_4, the algebraic expressions for F_3 and F_4 are not same. Therefore, the algebraic expression of a given Boolean function is not unique. In other words, it is possible to find more than one algebraic expression that specify the same Boolean function. As a matter of fact, the manipulation of Boolean algebra is applied mostly to the problem of finding simpler expression for the same function. The *simplification* of F_3 to F_4 can be achieved as below:

$$\begin{aligned}
F_3 &= x'y'z + x'yz + xy' \\
&= x'zy' + x'zy + xy' && \text{by postulate:} \quad 3(b) \\
&= x'z(y' + y) + xy' && 4(a) \\
&= x'z(y + y') + xy' && 3(a) \\
&= x'z \cdot 1 + xy' && 5(a) \\
&= x'z + xy' && 2(b) \\
&= xy' + x'z && 3(a) \\
&= F_4
\end{aligned}$$

Simplification of Boolean function will be discussed in detail in Chapter 5.

3.4.4 Logic Diagram

A Boolean function may be transformed from an algebraic expression into a *logic diagram* composed of AND, OR, and NOT gates. The implementation of the four Boolean functions introduced in the previous discussion is shown in Figure 3.1. The logic diagram includes a NOT gate for every variable present in its complement form. There is an AND gate for each AND term in the expression, and an OR gate is used to combine two or more AND terms. From the diagram, it is obvious that the implementation of F_4 requires fewer gates and fewer inputs than F_3. Since F_3 and F_4 are equal Boolean functions, it is more economical to implement the F_4 form than the F_3 form. To find simpler circuits, one must know how to manipulate Boolean functions to obtain equal and simpler expressions.

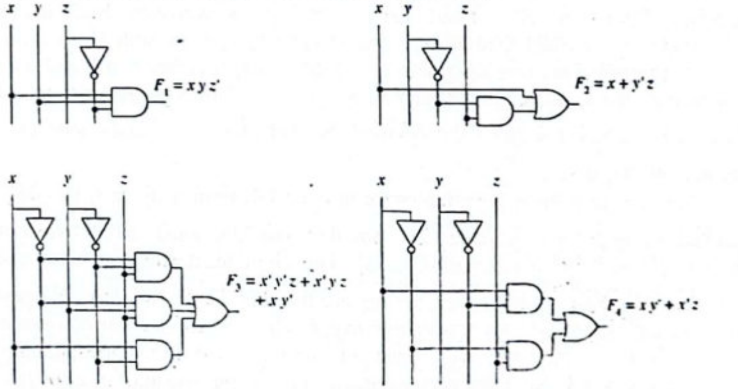


Figure 3.1 Implementation of Boolean functions with gates.

3.4.5 Complement of Boolean Function

The complement of a Boolean function F is F' and is obtained from an interchange of 0s for 1s and 1s for 0s in the value of F. The complement of a Boolean function may be derived algebraically through DeMorgan's theorem. For example, the complement of the functions $F_1 = x'yz' + x'y'z'$ and $F_2 = x(y'z' + yz)$ can be found as follows:

$$
\begin{aligned}
F_1' &= (x'yz' + x'y'z)' \\
&= (x'yz')'(x'y'z)' \\
&= (x + y' + z)(x + y + z') \\
F_2' &= (x(y'z' + yz))' \\
&= x' + (y'z' + yz)' \\
&= x' + (y'z')'(yz)' \\
&= x' + (y + z)(y' + z')
\end{aligned}
$$

3.5 Forms of Boolean Functions

3.5.1 Minterms and Maxterms

A binary variable may appear either in its normal form (x) or in its complement form (x'). Now consider two binary variables x and y combined with an AND operation. Since each variable may appear in either form, there are four possible combinations: $x'y'$, $x'y$, xy', and xy. Each of these four AND terms is called a *minterm* or a *standard product term* (or *standard product*). In a similar manner, n variables can be combined to form 2^n minterms. The 2^n different minterms may be determined by a method similar to the one shown in Table 3.6 for three variables. The binary numbers from 0 to $2^n - 1$ are listed under the n variables. Each minterm is obtained from an AND term of the n variables, with each variable being primed (or complemented) if the corresponding bit of the binary number is a 0 and unprimed (or normal) if a 1. A symbol for each minterm is also shown in the table and is of the form m_j, where j denotes the decimal equivalent of the binary number of the minterm designated.

In a similar fashion, n variables forming an OR term, with each variable being primed or unprimed, provide 2^n possible combinations, called *maxterms* or *standard sum terms* (or *standard sums*). The eight maxterms for three variables, together with their symbolic designation, are listed in Table 3.6. Any 2^n maxterms for n variables may be determined similarly. Each maxterm is obtained from an OR term of the n variables, with each variable being unprimed if the corresponding bit is 0 and primed if a 1. Note that *each maxterm is the complement of its corresponding minterm and vice versa.*

Table 3.6 Minterm and maxterm for three binary variables.

Variables	Minterms		Maxterms	
$x\ y\ z$	Term	Designation	Term	Designation
0 0 0	$x'y'z'$	m_0	$x + y + z$	M_0
0 0 1	$x'y'z$	m_1	$x + y + z'$	M_1
0 1 0	$x'yz'$	m_2	$x + y' + z$	M_2
0 1 1	$x'yz$	m_3	$x + y' + z'$	M_3
1 0 0	$xy'z'$	m_4	$x' + y + z$	M_4
1 0 1	$xy'z$	m_5	$x' + y + z'$	M_5
1 1 0	xyz'	m_6	$x' + y' + z$	M_6
1 1 1	xyz	m_7	$x' + y' + z'$	M_7

3.5.2 Canonical Forms

Boolean functions expressed in terms of minterms or maxterms are said to be in canonical form, because these representations are unique for the given functions. There are two canonical forms of Boolean functions: *canonical sum of products* (CSOP) and *canonical product of sums (CPOS).*

Canonical Sum of Products

A Boolean function may be expressed algebraically from a given truth table by forming a minterm for each combination of the variables which produces a 1 in the function, and then taking the OR of all these terms. For example, the function f_1 in Table 3.7 is determined by expressing the combinations 001, 100, and 111 as $x'y'z$, $xy'z'$, and xyz, respectively. Since each one of these minterms results in $f_1 = 1$, we should have:

$$f_1 = x'y'z + xy'z' + xyz = m_1 + m_4 + m_7.$$

Similarly, it may be easily verified that:

$$f_2 = x'yz + xy'z + xyz' + xyz = m_3 + m_5 + m_6 + m_7.$$

Table 3.7 Arbitrary functions of three variables.

$x\ y\ z$	Function f_1	Function f_2
0 0 0	0	0
0 0 1	1	0
0 1 0	0	0
0 1 1	0	1
1 0 0	1	0
1 0 1	0	1
1 1 0	0	1
1 1 1	1	1

These examples demonstrate an important property of Boolean algebra: *Any Boolean function can be expressed as a sum of minterms* (by "sum" is meant the ORing of terms). As these expressions are unique for the given Boolean functions, they are called *canonical sum of products (CSOP)* expressions. They are also called *sum of minterms* expressions.

Short Notation of Canonical Sum of Products Function

It is sometimes convenient to express the Boolean function, when in its canonical sum of products form, in the following short notation:

$$f_1(x,y,z)=\sum(1,4,7)$$
$$f_2(x,y,z)=\sum(3,5,6,7)$$

The summation symbol, $\sum$, stands for the ORing of terms; the numbers following it are the minterms of the function. The letters in parenthesis following f_1 or f_2 form a list of the variables in the order taken when the minterm is converted to an AND term.

For n binary variables, one can obtain 2^n distinct minterms, and any Boolean function can be expressed as a sum of minterms. The minterms whose sum defines the Boolean function are those that give the 1s of the function in a truth table. Since the function can be either 0 or 1 for each minterm, and since there are 2^n minterms, one can calculate the number of possible functions that can be formed with n variables to be 2^{2^n}.

Canonical Product of Sums

Now consider the complement of a Boolean function. It may be read from the truth table by forming a minterm for each combination that produces a 0 in the function and then ORing those terms. From Table 3.7, the complement of f_1 is read as:

$$f_1' = x'y'z' + x'yz' + x'yz + xy'z + xyz'$$

If we take the complement of f_1', we obtain the function f_1:

$$\begin{aligned} f_1 &= (x'y'z' + x'yz' + x'yz + xy'z + xyz')' \\ &= (x+y+z)(x+y'+z)(x+y'+z')(x'+y+z')(x'+y'+z) \\ &= M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6 \end{aligned}$$

Similarly, from Table 3.7, it is possible to read the expression for f_2:

$$\begin{aligned} f_2 &= (x+y+z)(x+y+z')(x+y'+z)(x'+y+z) \\ &= M_0 \cdot M_1 \cdot M_2 \cdot M_4 \end{aligned}$$

These examples demonstrate a second important property of Boolean algebra: *Any Boolean function can be expressed as a product of maxterms* (by "product" is meant the ANDing of terms). The procedure for obtaining the product of maxterms directly from the truth table is as follows. Form a maxterm for each combination of the variables which produces a 0 in the function, and then form the AND of all these maxterms. Each of the 2^{2^n} functions of n binary variables can be expressed as a product of maxterms.

As these expressions are also unique for the given Boolean functions, they are called *canonical product of sums (CPOS)* expression. They are also called *product of maxterms* expression.

Short Notation of Canonical Product of Sums Function

A convenient way to express canonical product of sums expression is as follows:

$$f_1(x,y,z)=\prod(0,2,3,5,6)$$
$$f_2(x,y,z)=\prod(0,1,2,4)$$

The product symbol, $\prod$, denotes the ANDing of maxterms; the number following it are the maxterms of the function.

Conversion between Canonical Forms

The complement of a function expressed as the canonical sum of products equals the sum of minterms missing from the original function. This is because the original function is expressed by those minterms that make the function equal to 1, while its complement is a 1 for those minterms that the function is a 0. As an example, consider the function:

$$F(A,B,C)=\sum(1,4,5,6,7)$$

This has a complement that can be expressed as:

$$F'(A,B,C)=\sum(0,2,3)=m_0+m_2+m_3$$

Now, if we take the complement of F' by DeMorgan's theorem, we obtain F in a different form:

$$\begin{aligned} F(A,B,C) &= (m_0+m_2+m_3)' = m_0' \cdot m_2' \cdot m_3' = M_0 \cdot M_2 \cdot M_3 \\ &= \prod(0,2,3) \end{aligned}$$

The last conversion follows from the definition of minterms and maxterms as shown in Table 3.6. From the table, it is clear that the following relation holds true:

$$m_j' = M_j$$

That is, the maxterm with subscript j is a complement of the minterm with the same subscript j, and vice versa.

The last example demonstrates the conversion between a function expressed in canonical sum of products and its equivalent in canonical product of sums. A similar argument will show that the conversion between the canonical product of sums and the canonical sum of products is similar. We now state a general conversion procedure. To convert from one canonical from to another, interchange the symbols $\sum$ and $\prod$ and list those numbers missing from the original form. As another example, the function:

$$F(x, y, z) = \prod(0,2,4,5)$$

is expressed in the canonical product of sums form. Its conversion to canonical sum of products is:

$$F(x, y, z) = \sum(1,3,6,7)$$

Note that, in order to find the missing terms, one must realize that the total number of minterms or maxterm is 2^n, where n is the number of binary variables in the function.

3.5.3 Noncanonical Forms

The two canonical forms of Boolean algebra are basic forms that one obtains from reading a function from the truth table. In these forms, each minterm or maxterm contains n literals, where n is the number of variables. A *literal* is a primed (complemented) or unprimed (uncomplemented or normal) variable.

Another way to express Boolean functions is in *noncanonical form*. In this configuration, the terms that form the function may contain one, two or any number of literals. There are two types of noncanonical forms: *sum of products (SOP)* and *product of sums (POS)*.

Sum of Products Form

The *sum of products (SOP)* is a Boolean expression containing AND terms, called *product terms*, of one or more literals each. The *sum* denotes the ORing of these terms. An example of a function expressed in sum of products form is:

$$F_1(x, y, z) = y' + xy + x'yz'$$

The expression has three product terms of one, two, and three literals each, respectively. Their sum is in effect an OR operation.

Conversion of Sum of Products into Canonical Sum of Products

It is sometimes needed to convert the Boolean function expressed as sum of products form into its canonical sum of products or sum of minterms form. It can be done by first inspecting each AND term to see if it contains all the variables. If it misses one or more variables, it is ANDed with an expression such as $x + x'$, where x is one of the missing variables. The following example clarifies the procedure:

$$F(A, B, C) = A + B'C$$

$$= A(B + B')(C + C') + (A + A')B'C$$
$$= (AB + AB')(C + C') + AB'C + A'B'C$$
$$= ABC + ABC' + AB'C + AB'C' + AB'C + A'B'C$$
$$= A'B'C + AB'C' + (AB'C + AB'C) + ABC' + ABC$$
$$= A'B'C + AB'C' + AB'C + ABC' + ABC$$
$$= m_1 + m_4 + m_5 + m_6 + m_7 = \sum(1,4,5,6,7)$$

Product of Sums Form

A *product of sums (POS)* is a Boolean expression containing OR terms, called *sum terms*. Each term may have any number of literals. The *product* denotes the ANDing of these terms. An example of a function expressed in product of sums is:

$$F_2(x, y, z) = x(y' + z)(x' + y + z')$$

This expression has three terms of one, two, and three literals each. The product is an AND operation.

Conversion of Product of Sums into Canonical Product of Sums

To convert the Boolean function expressed as product of sums form into a canonical product of sums or product of maxterms form, an OR term having any missing variable x in it is ORed with xx'. This procedure is clarified by the following example:

$$F(x, y, z) = (x' + y)(x + z)(y + z)$$
$$= (x' + y + zz')(x + yy' + z)(xx' + y + z)$$
$$= (x' + y + z)(x' + y + z')(x + y + z)(x + y' + z)(x + y + z)(x' + y + z)$$
$$= (x + y + z)(x + y + z)(x + y' + z)(x' + y + z)(x' + y + z)(x' + y + z')$$
$$= (x + y + z)(x + y' + z)(x' + y + z)(x' + y + z')$$
$$= M_0 \cdot M_2 \cdot M_4 \cdot M_5 = \prod(0,2,4,5)$$

3.5.4 Nonstandard Form

The canonical and noncanonical forms of Boolean functions discussed earlier are the standard forms of Boolean functions. However, a Boolean function may be expressed in a nonstandard form. For example, the function:

$$F_3(A, B, C, D) = (AB + CD)(A'B' + C'D')$$

is neither in sum of products nor in product of sums.

Conversion of Nonstandard Form into Standard Form

A Boolean function expressed in nonstandard form can be changed to a standard from by using the distributive law $x(y + z) = xy + xz$ or $x + yz = (x + y)(x + z)$. This procedure is clarified by the following example:

$$
\begin{aligned}
F_3(A,B,C,D) &= (AB+CD)(A'B'+C'D') \\
&= (AB+CD)A'B'+(AB+CD)C'D' \\
&= ABA'B'+CDA'B'+ABC'D'+CDC'D' \\
&= A'B'CD+ABC'D'
\end{aligned}
$$

Again,

$$F_3(A,B,C,D) = (AB+CD)(A'B'+C'D')$$

$$= (AB+C)(AB+D)(A'B'+C')(A'B'+D')$$

$$= (A+C)(B+C)(A+D)(B+D)(A'+C')(B'+C')(A'+D')(B'+D')$$

3.6 Incompletely Specified Boolean Functions

In Boolean functions so far discussed, the value for the function is specified for all 2^n possible input combinations of n variables. These Boolean functions are said to be *completely specified* Boolean functions. Sometimes, however, for some input combinations the output may not be specified. It means that the outputs for these particular combinations are immaterial, as far as the operation of the corresponding digital circuit is concerned. This is also possible when certain input combinations can never occur due to some constraints. For example, in BCD code the four-bit combinations from 0000 to 1001 are assigned to decimal digits 0 to 9, and the combinations from 1010 to 1111 are unused. The outputs for these combinations are immaterial for the operation of the BCD code converter, since these combinations will never occur. Therefore, we don't care about the state of the output for these input combinations. Such situations are indicated on the truth table by writing x (or sometimes d) as the functional value, rather than 0 or 1. Such conditions are referred to as *don't care conditions* and the Boolean functions which include don't care conditions are said to be *incompletely specified* Boolean functions. For example, consider the Boolean function described by the truth table in Table 3.8. The function is an incompletely specified function and can be written as

$$f(x,y,z) = \sum(0,4,7) + \sum\nolimits_{d.c.}(2,6)$$

or

$$f(x,y,z) = \prod(1,3,5) \cdot \prod\nolimits_{d.c.}(2,6)$$

Table 3.8 Truth table of an incompletely specified Boolean function.

$x\ y\ z$	f
0 0 0	1
0 0 1	0
0 1 0	x
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	x
1 1 1	1

Exercises

3.1 Demonstrate by means of truth tables the validity of the following theorems of Boolean algebra.

(a) The associative laws.

(b) DeMorgan's theorems for three variables.

3.2 Using DeMorgan's theorems, convert the following expressions to one that has only single variable inversion.

(a) $F = ((A+B)C')'$ $\quad F = (RS'T + Q')'$

3.3 Obtain the truth table of the functions.

(a) $F = XY + XY' + Y'Z$ (b) $F = (X + Y')(X' + Y)(Y' + Z)$

3.4 Find the complement of the following Boolean functions.

(a) $F = (BC' + A'D)(AB' + CD')$

(b) $F = B'D + A'BC' + ACD + A'BC$

(c) $F = ((AB)'A)((AB)'B)$

(d) $F = AB' + C'D'$

3.5 Implement the following functions with logic gates:

(a) $F = xy + xy' + y'z$

(b) $F = (x + y')(x' + y)(y' + z)$

(c) $F = y(wz' + wz) + xy$

(d) $F = ((yz)' + w)' + w + yz + wx$

(e) $F = A'BC(A + D')$

(f) $F = AC + BC' + A'BC$

3.6 Expand the following function to their canonical forms.

(a) $F(A,B,C,D) = BC + AC' + AB + BCD$

(b) $F(A,B,C,D) = (A + C + D)(A + C + D')$
$(A + C' + D)(A + B')$

3.7 Express the following functions in a canonical sum of products and a canonical product of sums form.

(a) $F(A,B,C,D) = D(A' + B) + B'D$

(b) $F(w,x,y,z) = y'z + wxy' + wxz' + w'x'z$

(c) $F(A,B,C,D) = (A + B' + C)(A + B')(A + C' + D')$
$(A' + B + C + D')(B + C' + D')$

(d) $F(A,B,C) = (A' + B)(B' + C)$

(e) $F(x,y,z) = 1$

(f) $F(x,y,z) = (xy + z)(y + xz)$

3.8 Convert the following to the other canonical form.

(a) $F(x, y, z) = \sum(1,3,7)$

(b) $F(A, B, C, D) = \sum(0,2,6,11,13,14)$

(c) $F(x, y, z) = \prod(0,3,6,7)$

(d) $F(A, B, C, D) = \prod(0,1,2,3,4,6,12)$

3.9 What is the difference between canonical form and noncanonical form? Which form is preferable when implementing a Boolean function with gates? Which form is obtained when reading a function from a truth table?

3.10 The sum of all minterms of a Boolean function of n variables is 1. Prove the statement for $n = 3$.

3.11 The product of all maxterms of a Boolean function of n variables is 0. Prove the statement for $n = 3$.

3.12 Write the canonical sum of products and canonical product of sums function using short notation for the function described in the following truth table.

$A\,B\,C\,D$	F
0 0 0 0	0
0 0 0 1	0
0 0 1 0	1
0 0 1 1	1
0 1 0 0	0
0 1 0 1	x
0 1 1 0	0
0 1 1 1	1
1 0 0 0	x
1 0 0 1	x
1 0 1 0	1
1 0 1 1	1
1 1 0 0	0
1 1 0 1	0
1 1 1 0	1
1 1 1 1	0

Further Reading

1. E. J. McCluskey, *Logic Design Proncíples,* Prentice-Hall, Inc., 1986.
2. Zvi Kohavi, *Switching and Finite Automata Theory*, 2nd ed., McGraw-Hill Higher Education, 1990.
3. Richard S. Sandige, *Modern Digital Design*, McGraw-Hill Publishing Company, 1990.

4. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
5. John P. Uyemura, *Digital Systems Design: An Integrated Approach*, Brooks/Cole Publishing Company, 2000.
6. Ronald J. Tocci and Neal S. Widmer, *Digital Systems: Principles and Applications*, 8th ed., Prentice-Hall, Inc., 2001.
7. V. K. Jain, *Switching Theory and Digital Electronics*, Khanna Publishers, India, 2002.
8. Samuel C. Lee, *Digital Circuits and Logic Design*, Prentice-Hall, Inc., 2002.

CHAPTER 4
LOGIC GATES

4.1 Introduction

Logic gates are the basic building blocks of digital systems. *Logic gates* are electronic circuits that perform the operations of Boolean algebra. In this chapter, we shall describe logic gates and their operations using Boolean algebra. Integrated circuit (IC) logic families will also be introduced.

4.2 Logic Operations

When the binary operators AND and OR are placed between two variables x and y, they form two Boolean functions $x \cdot y$ and $x + y$, respectively. It was stated in the previous chapter that there are 2^{2^n} functions for n binary variables. For two variables, $n = 2$ and the number of possible Boolean functions is 16. Therefore, the AND and OR functions are only two of a total of 16 possible functions formed with two binary variables.

The truth tables for the 16 functions formed with two binary variables x and y are listed in Table 4.1. In this table, each of the 16 columns F_0 to F_{15} represents a truth table of one possible function for two given variables x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F.

Table 4.1 Truth tables for the 16 functions of two binary variables.

xy	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0 0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0 1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1 0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1 1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The 16 functions listed in truth table form can be expressed algebraically by means of Boolean expression. This is shown in the first column of Table 4.2. The Boolean expressions listed are in their simplified form. Although each function can be expressed in terms of the Boolean operators AND, OR, and NOT, there is no reason one cannot assign special operator symbols for expressing the other functions. Such operator symbols are listed in the second column of Table 4.2. However, all the new symbols shown, except for the EXOR symbol ⊕, are not in common use by digital designers.

Table 4.2 Boolean expressions for the 16 functions of two variables.

Boolean functions	Operator symbol	Name	Comment
$F_0 = 0$		Null	Binary constant 0
$F_1 = xy$	$x \cdot y$	AND	*x* and *y*
$F_2 = xy'$	x / y	Inhibition	*x* but not *y*
$F_3 = x$		Transfer	*x*
$F_4 = x'y$	y / x	Inhibition	*y* but not *x*
$F_5 = y$		Transfer	*y*
$F_6 = xy' + x'y$	$x \oplus y$	EXOR	*x* or *y* but not both
$F_7 = x + y$	$x + y$	OR	*x* or *y*
$F_8 = (x + y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$x \otimes y$	EXNOR (Equivalence)	*x* equals *y*
$F_{10} = y'$	y'	NOT (Inversion)	Not *y*
$F_{11} = x + y'$	$x \subset y$	Implication	If *y* then *x*
$F_{12} = x'$	x'	NOT (Inversion)	Not *x*
$F_{13} = x' + y$	$x \supset y$	Implication	If *x* then *y*
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$		Identity	Binary constant 1

Each of the functions in Table 4.2 is listed with an accompanying name and a comment that explains the function in some way. The 16 functions listed can be subdivided into three categories:

1. Two functions (F_0 and F_{15}) that produce a constant 0 or 1.
2. Four functions (F_3, F_5, F_{10}, and F_{12}) with unary operators NOT and transfer.
3. Ten functions (F_1, F_2, F_4, F_6, F_7, F_8, F_9, F_{11}, F_{13}, and F_{14}) with binary operators that define eight different operations AND (F_1), OR (F_7), NAND (F_{14}), NOR (F_8), EXOR (F_6), EXNOR (F_9), inhibition (F_2 and F_4), and implication (F_{11} and F_{13}).

Any binary function can be equal to 0 or 1. The NOT function produces the complement of each of the binary variables. A function that is equal to an input variable has been given the name *transfer*, because the variable x or y is transformed through the gate that forms the function without changing its value. Of the eight binary operators, two (inhibition and implication) are used by logicians but are seldom used in digital logic. The AND and OR operators have been mentioned in conjunction with Boolean algebra. The other four operators such as NAND, NOR, EXOR, and EXNOR are extensively used in the design of digital systems.

The NOR function is the complement of the OR function and its name is an abbreviation of *not*-OR. Similarly, NAND is the complement of AND and is an abbreviation of *not*-AND. The EXOR is an abbreviation of *exclusive*-OR and is similar to OR but excludes the combination of *both* x and y being equal to 1. The EXNOR or *equivalence* is a function that is 1 when the two binary variables are equal, i.e., when both are 0 or both are 1. EXNOR is an abbreviation of *exclusive-*OR-*not*. The EXOR and EXNOR are the complements of each other. This can be easily verified by inspecting Table 4.1. The truth table for the EXOR is F_6 and for the EXNOR is F_9, and these two functions are the complements of each other.

4.3 Logic Gates

Since Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement a Boolean function with these types of gates. The possibility of constructing gates for the other logic operations is of practical interest. Factors to be weighed when considering the construction of other types of logic gates are

(1) the feasibility and economy of producing the gate with physical electronic components,
(2) the possibility of extending the gate to more than two inputs,
(3) the basic properties of the binary operator such as commutativity and associativity, and
(4) the ability of the gate to implement Boolean functions alone or in conjunction with other gates.

Of the 16 functions defined in Table 4.2 two are equal to a constant and four others are repeated twice. There are only ten functions left to be considered as candidates for logic gates. Two, inhibition and implication, are not commutative or associative and thus are impractical to use as standard logic gates. The other eight: transfer, NOT, AND, OR, NAND, NOR, EXOR, and EXNOR, are used as standard gates in digital design. The graphic symbols and truth tables of the eight gates are shown in Figure 4.1. Each gate has one or two binary input variables designated by x and y and one binary output variable designated by F.

A buffer produces the *transfer* function but does not produce any particular logic operations, since the binary value of the output is equal to the binary value of the input. This is used merely for power amplification of the signal. The NOT gate inverts the logic sense of a binary variable. It produces the complement function. The triangle symbol designates a buffer and the small circle in the output of the graphic buffer symbol designates the logic complement.

The AND and OR gates were defined in Figure 1.4. The NAND function is the complement of the AND function, as indicated by a graphic symbol which consists of an AND graphic symbol followed by a small circle. The NOR function is the complement of the OR function and uses an OR graphic symbol followed by a small circle. The NAND and NOR gates are extensively used as standard logic gates and are in fact far more popular than the AND and OR gates. This is because NAND and NOR gates are easily constructed with transistor circuits and because these gates are *universal* gates, that is, Boolean functions can be easily implemented with them.

The EXOR gate has a graphic symbol similar to that of the OR gate, except for the additional curved line on the input side. The EXNOR gate is the complement of the EXOR gate, as indicated by the small circle on the output side of the graphic symbol. The EXOR gates are finding more and more use in digital logic design.

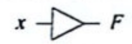
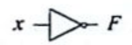
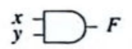
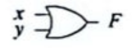
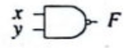
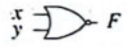
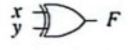
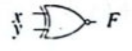
Name	Graphic symbol	Algebraic function	Truth table
Buffer (transfer)		$F = x$	x F 0 0 1 1
NOT (inverter)		$F = x'$	x F 0 1 1 0
AND		$F = xy$	x y F 0 0 0 0 1 0 1 0 0 1 1 1
OR		$F = x + y$	x y F 0 0 0 0 1 1 1 0 1 1 1 1
NAND		$F = (xy)'$	x y F 0 0 1 0 1 1 1 0 1 1 1 0
NOR		$F = (x + y)'$	x y F 0 0 1 0 1 0 1 0 0 1 1 0
EXOR		$F = xy' + x'y$ $= x \oplus y$	x y F 0 0 0 0 1 1 1 0 1 1 1 0
EXNOR (equivalence)		$F = xy + x'y'$ $= x \otimes y$	x y F 0 0 1 0 1 0 1 0 0 1 1 1

Figure 4.1 Digital logic gates.

Extension to Multiple Inputs

The gates shown in Figure 4.1, except for the buffer and NOT, can be extended to have more than two inputs. A gate can be extended to have multiple inputs if the binary operation it represents is commutative and associative. The AND and OR operations, defined in Boolean algebra, possess these two properties. For the AND and OR functions we have:

$$\left.\begin{aligned} x\cdot y &= y\cdot x \\ x+y &= y+x \end{aligned}\right\} \quad \text{commutative}$$

and

$$\left.\begin{aligned} (x\cdot y)\cdot z = x\cdot(y\cdot z) &= x\cdot y\cdot z \\ (x+y)+z = x+(y+z) &= x+y+z \end{aligned}\right\} \quad \text{associative}$$

which indicates that the gate inputs can be interchanged and that the AND and OR functions can be extended to three or more variables. Multiple-input AND and OR gates were introduced in Figure 1.4.

The construction of three-input AND and OR gates by cascading two-input gates is shown in Figure 4.2(a) and (c) respectively. Graphically, they can be represented with a simple three-input gate as shown in Figure 4.2(b) and (d) respectively.

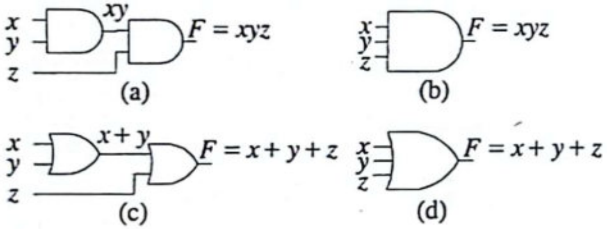


Figure 4.2 Three-input AND and OR gates – (a) 3-input AND gate using 2-input AND gates, (b) graphic symbol of 3-input AND gate, (c) 3-input OR gate using 2-input OR gates, and (d) graphic symbol of 3-input OR gate.

The NAND and NOR functions are commutative and their gates can be extended to have more than two inputs, provided the definitions of the operation is slightly modified. The difficulty is that the NAND and NOR operations are not associative, i.e., $(x\uparrow y)\uparrow z \neq x\uparrow(y\uparrow z)$ and $(x\downarrow y)\downarrow z \neq x\downarrow(y\downarrow z)$, as shown below:

$$(x\uparrow y)\uparrow z = ((xy)'\cdot z)' = (xy)+z' = (x+z')(y+z')$$
$$x\uparrow(y\uparrow z) = (x\cdot(yz)')' = x'+(yz) = (x'+y)(x'+z)$$

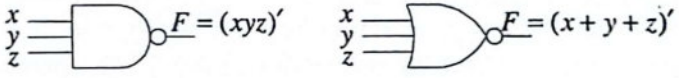
and

$$(x\downarrow y)\downarrow z = ((x+y)'+z)' = (x+y)\cdot z' = xz'+yz'$$
$$x\downarrow(y\downarrow z) = (x+(y+z)')' = x'\cdot(y+z) = x'y+x'z$$

To overcome this difficulty, we define the multiple-input NAND and NOR gates as complemented AND and OR gates respectively. Thus, by definition, we have:

$$x\uparrow y\uparrow z = (xyz)'$$
$$x\downarrow y\downarrow z = (x+y+z)'$$

The graphic symbols for the three-input NAND and NOR gates are shown in Figure 4.3.



(a) 3-input NAND gate (b) 3-input NOR gate

Figure 4.3 Multiple-input NAND and NOR gates.

As the NAND and NOR operations are not associative, in writing cascaded NAND and NOR operations, one must use the correct parentheses to signify the proper sequence of the gates. To demonstrate this, consider the circuit of Figure 4.4. The Boolean function for the circuit of Figure 4.4(a) must be written as:

$$F = ((ABC)'(DE)')' = ABC + DE$$

and the Boolean function for the circuit of Figure 4.4(b) must be written as:

$$F = ((A+B+C)'+(D+E)')' = (A+B+C)(D+E)$$

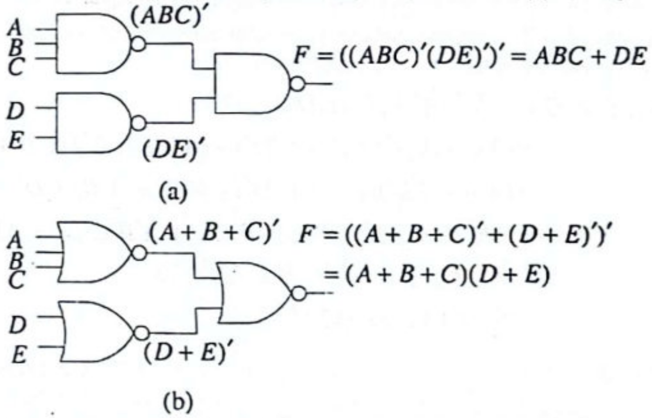


Figure 4.4 (a) Cascaded NAND gates and (b) cascaded NOR gates.

The EXOR and EXNOR functions are both commutative and associative, that is,

$$\left.\begin{aligned}A \oplus B &= B \oplus A\\ A \otimes B &= B \otimes A\end{aligned}\right\} \text{commutative}$$

and

$$\left.\begin{aligned}(A \oplus B) \oplus C &= A \oplus (B \oplus C) = A \oplus B \oplus C\\ (A \otimes B) \otimes C &= A \otimes (B \otimes C) = A \otimes B \otimes C\end{aligned}\right\} \text{associative}$$

Because of these two properties, we can extend EXOR and EXNOR gates to more than two inputs. However, multiple-input EXOR and EXNOR gates are uncommon from the hardware standpoint. In fact, even a two-input gate is usually constructed with other types of gates. Moreover, the definition of these functions must be modified when extended to more than two variables. The EXOR is an *odd* function, i.e., it is equal to 1 if the input variables have an odd number of 1s. An n-variable EXOR expression is equal to the Boolean function with $2^n/2$ minterms whose equivalent binary numbers have an odd number of 1s. For example,

$$\begin{aligned}A \oplus B \oplus C \oplus D &= (AB' + A'B) \oplus (CD' + C'D)\\ &= (AB' + A'B)(CD' + C'D)' + (AB' + A'B)'(CD' + C'D)\\ &= (AB' + A'B)(CD + C'D') + (AB + A'B')(CD' + C'D)\\ &= AB'CD + AB'C'D' + A'BCD + A'BC'D' + ABCD'\\ &\quad + ABC'D + A'B'CD' + A'B'C'D\\ &= \sum(1,2,4,7,8,11,13,14)\end{aligned}$$

The EXNOR is an *even* function, i.e., it is equal to 1 if the input variables have an even number of 0s. An n-variable EXNOR expression is equal to the Boolean function with $2^n/2$ minterms, whose equivalent binary numbers have an even number of 0s. For example,

$$\begin{aligned}A \otimes B \otimes C \otimes D &= (AB + A'B') \otimes (CD + C'D')\\ &= (AB + A'B')(CD + C'D') + (AB + A'B')'(CD + C'D')'\\ &= (AB + A'B')(CD + C'D') + (AB' + A'B)(CD' + C'D)\\ &= ABCD + ABC'D' + A'B'CD + A'B'C'D' + AB'CD'\\ &\quad + AB'C'D + A'BC'D + AB'C'D\\ &= \sum(0,3,5,6,9,10,12,15)\end{aligned}$$

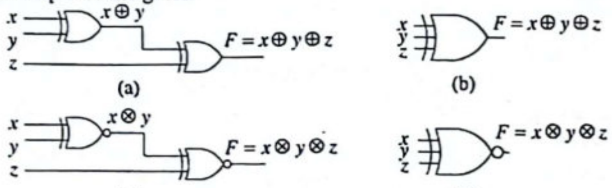
When the number of variables in a function is even, EXOR expression and EXNOR expression are complement of each other. This is evident from the above discussion, because $A \oplus B \oplus C \oplus D = \sum(1,2,4,7,8,11,13,14)$ and $A \otimes B \otimes C \otimes D = \sum(0,3,5,6,9,10,12,15)$. When the number of variables in a function is odd, the minterms with an even number of 0s are the same as the minterms with an odd number of 1s. Therefore, in this case, EXOR expression is equal to EXNOR expression. For example,

$$\begin{aligned}A \oplus B \oplus C &= A \oplus (BC' + B'C)\\ &= A(BC' + B'C)' + A'(BC' + B'C)\\ &= A(BC + B'C') + A'(BC' + B'C)\\ &= ABC + AB'C' + A'BC' + A'B'C\\ &= \sum(1,2,4,7)\end{aligned}$$

and

$$\begin{aligned}A \otimes B \otimes C &= A \otimes (BC + B'C')\\ &= A(BC + B'C') + A'(BC + B'C')'\\ &= A(BC + B'C') + A'(BC' + B'C)\\ &= ABC + AB'C' + A'BC' + A'B'C\\ &= \sum(1,2,4,7)\end{aligned}$$

The construction of three-input EXOR and EXNOR gates is shown in Figure 4.5. They are normally implemented by cascading two-input gates as shown in Figure 4.5(a) and (c). Graphically, they can be represented with a single three-input gate as shown in Figure 4.5(b) and (d). The truth tables in Figure 4.5(e) clearly indicate the output of the gates.


x
y
z
$x \oplus y$
$F = x \oplus y \oplus z$
(a)
x
y
z
$F = x \oplus y \oplus z$
(b)
x
y
z
$x \otimes y$
$F = x \otimes y \otimes z$
(c)
x
y
z
$F = x \otimes y \otimes z$
(d)

x	y	z	$F = x \oplus y \oplus z$	$F = x \otimes y \otimes z$
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	0
1	0	0	1	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

(e)

Figure 4.5 Three-input EXOR and EXNOR gates – (a) 3-input EXOR gate using 2-input EXOR gates, (b) graphic symbol of 3-input EXOR gate, (c) 3-input EXNOR gate using 2-input EXNOR gates, (d) graphic symbol of 3-input EXNOR gate, and (e) truth table of 3-input EXOR and EXNOR gates.

4.4 Universality of NAND and NOR Gates

All Boolean expressions consist of various combinations of the basic operations of AND, OR, and NOT. Therefore, any expression can be implemented using combinations of AND gates, OR gates, and NOT gates. It is possible, however, to implement any logic expression using only NAND gates and no other type of gate. This is because NAND gates, in the proper combination, can be used to perform each of the Boolean operations AND, OR, and NOT. This is demonstrated in Figure 4.6. In Figure 4.6(a), we have a two-input NAND gate whose input are purposely connected together so that the variable A is applied to both. In this configuration, the NAND simply acts as NOT, since its output is $F = (A \cdot A)' = A'$. In Figure 4.6(b), we have two NAND gates connected so that the AND operation is performed. NAND gate 2 is used as a NOT gate to change $(AB)'$ to $((AB)')' = AB$, which is the desired AND function. In Figure 4.6(c), we have three NAND gates connected so that the OR operation is performed. Here NAND gates 1 and 2 are used as NOT gates to invert the inputs, so that the final output is $F = (A' \cdot B')' = A + B$, which is the desired OR function.

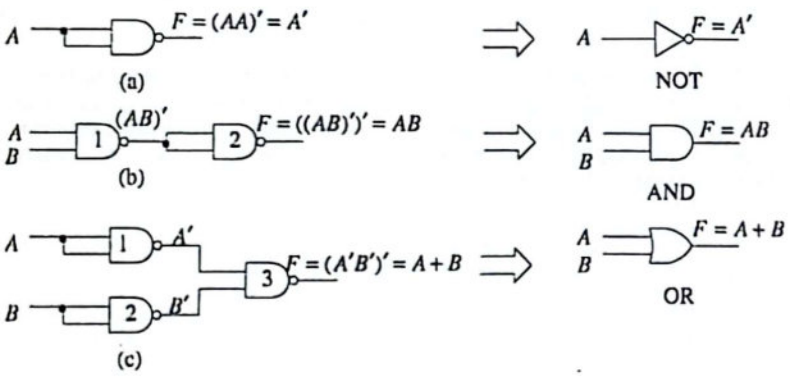


Figure 4.6 NAND gates can be used to implement (a) NOT, (b) AND, and (c) OR gates.

In a similar manner, it can be shown that NOR gates can be arranged to implement AND, OR, and NOT operations. This is illustrated in Figure 4.7. In Figure 4.7(a), NOR gate with its inputs connected together behaves as a NOT gate, since the output is $F = (A + A)' = A'$. In Figure 4.7(b), two NOR gates are arranged so that the OR operation is performed. NOR gate 2 is used as a NOT gate to change $(A + B)'$ to $((A + B)')' = A + B$, which is the desired OR function. In Figure 4.7(c), three NOR gates are used to implement AND operation. Here NOR gates 1 and 2 are used as NOT gates to invert the inputs, so that the final output is $F = (A' + B')' = AB$, which is the desired AND function.

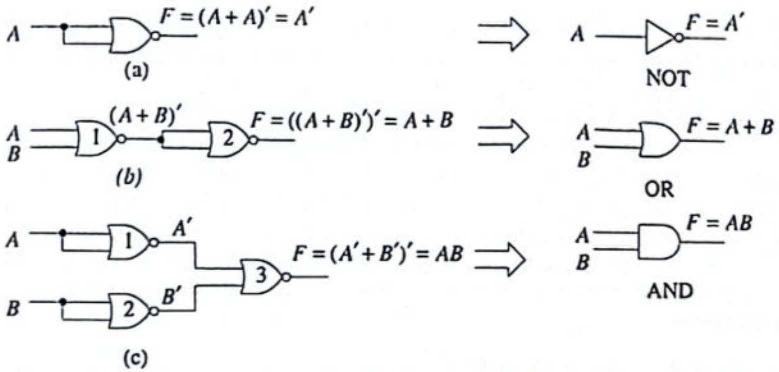
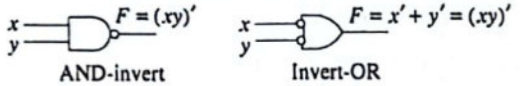


Figure 4.7 NOR gates can be used to implement (a) NOT, (b) OR, and (c) AND gates.

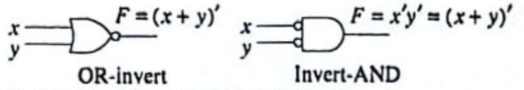
Since any of the Boolean operations can be implemented using only NAND gates, any logic circuit can be constructed using only NAND gates. The same is true for NOR gates. We shall discuss these topics in detail in Chapter 6.

4.5 Alternative Graphic Symbols for NAND, NOR, and NOT Gates

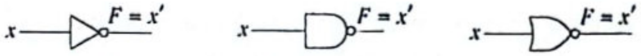
It is convenient in many cases to define two other graphic symbols for NAND and NOR gates. Two equivalent symbols for the NAND gate are shown in Figure 4.8(a). The AND-invert symbol has been defined previously and consists of an AND graphic symbol followed by a small circle. Instead, it is possible to represent a NAND gate by an OR graphic symbol preceded by small circles in all the inputs. The invert-OR symbol for the NAND gate follows from DeMorgan's theorem and from the convention that small circles denote complementation.



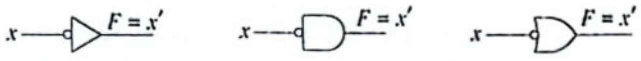
(a) Two graphic symbols for NAND gate



(b) Two graphic symbols for NOR gate



(c) Three graphic symbols for NOT gate



(d) Three alternate graphic symbols for NOT gate

Figure 4.8 Alternative graphic symbols for NAND, NOR, and NOT gates.

Similarly, there are two graphic symbols for the NOR gate as shown in Figure 4.8(b). The OR-invert is the conventional symbol. The invert-AND is a convenient alternative that utilizes DeMorgan's theorem and the convention that small circle in the inputs denote complementation.

A two-input NAND or NOR gate with its input connected together behaves like NOT gate. As a consequence, a NOT gate can be drawn in three different ways as shown in Figure 4.8(c). The small circles in all NOT symbols can be transferred to the input terminal without changing the logic of the gate as shown in Figure 4.8(d).

These alternative graphic symbols for NAND and NOR gates are very useful in NAND and NOR circuit implementation, which will be discussed in detail in Chapter 6.

4.6 EXOR and EXNOR Gates

4.6.1 Properties of EXOR and EXNOR Operations

The EXOR and EXNOR operations are finding more and more applications in digital systems design. We shall discuss EXOR circuits in detail in Chapter 6. However, some basic properties of EXOR and EXNOR operations are stated in the following theorems.

Theorems

1. (a) $X \oplus X = 0$ (b) $X \otimes X = 1$
2. (a) $X \oplus X' = 1$ (b) $X \otimes X' = 0$
3. (a) $1 \oplus X = X'$ (b) $0 \otimes X = X'$
4. (a) $X \oplus Y = X' \oplus Y'$ (b) $X \otimes Y = X' \otimes Y'$

The proofs of the above theorems are given below.

1. (a) $X \oplus X = XX' + X'X = 0$

   (b) $X \otimes X = XX + X'X' = X + X' = 1$

2. (a) $X \oplus X' = X(X')' + X'X' = X + X' = 1$

   (b) $X \otimes X' = XX' + X'(X')' = XX' + X'X = 0$

3. (a) $1 \oplus X = 1 \cdot X' + 1' \cdot X = 1 \cdot X' + 0 \cdot X = X'$

   (b) $0 \otimes X = 0 \cdot X + 0' \cdot X' = 0 \cdot X + 1 \cdot X' = X'$

4. (a) From 1(a), we have that $X' \otimes X = 0$. To prove $X \oplus Y = X' \oplus Y'$, we shall prove $(X \oplus Y) \oplus (X' \oplus Y') = 0$.

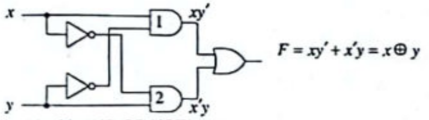
$$
\begin{aligned}
(X \oplus Y) \oplus (X' \oplus Y') &= X \oplus Y \oplus X' \oplus Y' && \text{using associative law}\\
&= X \oplus X' \oplus Y \oplus Y' && \text{using commutative law}\\
&= 1 \oplus 1 && \text{using theorem 2(a)}\\
&= 0 && \text{using theorem 1(a)}
\end{aligned}
$$

(b) From 1(b), we know that $X \otimes X = 1$. To prove $X \otimes Y = X' \otimes Y'$, we shall prove $(X \otimes Y) \otimes (X' \otimes Y') = 1$.

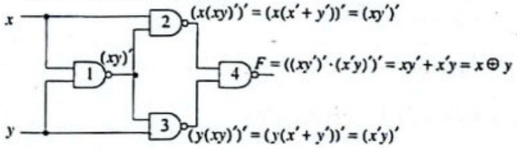
$$
\begin{aligned}
(X \otimes Y) \otimes (X' \otimes Y') &= X \otimes Y \otimes X' \otimes Y' && \text{using associative law}\\
&= X \otimes X' \otimes Y \otimes Y' && \text{using commutative law}\\
&= 0 \otimes 0 && \text{using theorem 2(b)}\\
&= 1 && \text{using theorem 1(b)}
\end{aligned}
$$

4.6.2 EXOR and EXNOR Gate Implementations

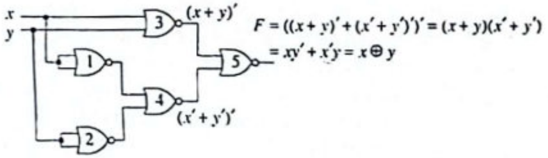
As discussed in section 4.3, a two-input EXOR gate is usually constructed with other types of gates. Implementation of a two-input EXOR gate is shown in Figure 4.9. In Figure 4.9(a), the output of AND gate 1 is xy' and the output of AND gate 2 is $x'y$. The final output of OR gate is $xy' + x'y$, which is the required EXOR expression. In Figure 4.9(b), the output of NAND gate 1 is $(xy)'$. The output of NAND gate 2 is $(x(xy)')' = (x(x' + y'))' = (xy')'$. The output of NAND gate 3 is $(y(xy)')' = (y(x' + y'))' = (x'y)'$. The final output of NAND gate 4 is $((xy')'(x'y)')' = xy' + x'y$, which is the required EXOR expression. In Figure 4.9(c), NOR gate 1 and 2 act as inverter. The output of NOR gate 3 is $(x + y)'$ and the output of NOR gate 4 is $(x' + y')'$. The final output of NOR gate 5 is $((x + y)' + (x' + y')')' = (x + y)(x' + y') = xy' + x'y$, which is the required EXOR expression.



(a) with AND-OR-NOT gates


x
2
(x(xy)')' = (x(x' + y'))' = (xy')'
(xy)'
1
4
F = ((xy')'·(x'y)')' = xy' + x'y = x ⊕ y
y
3
(y(xy)')' = (y(x' + y'))' = (x'y)'

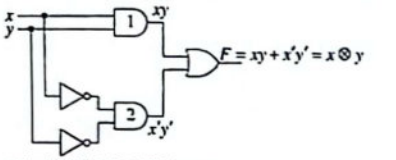
(b) with NAND gates



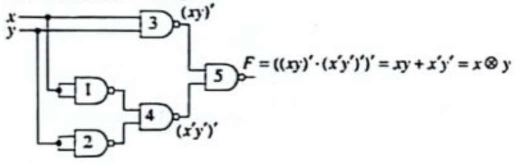
(c) with NOR gates

Figure 4.9 Implementation of EXOR gate with other gates.

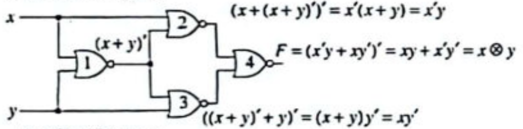
Implementation of a two-input EXNOR gate is shown in Figure 4.10. In Figure 4.10(a), the output of AND gate 1 is xy and the output of AND gate 2 is $x'y'$. The final output of OR gate is $xy+x'y'$, which is the required EXNOR expression. In Figure 4.10(b), NAND gate 1 and 2 act as inverter. The output of NAND gate 3 is $(xy)'$ and the output of NAND gate 4 is $(x'y')'$. The final output of NAND gate 5 is $((xy)'\cdot(x'y')')'=xy+x'y'$, which is the required EXNOR expression. In Figure 4.8(c), the output of NOR gate 1 is $(x+y)'$. The output of NOR gate 2 is $(x+(x+y)')'=x'(x+y)=x'y$. The output of NOR gate 3 is $((x+y)'+y)'=(x+y)y'=xy'$. The final output of NOR gate 4 is $(x'y+xy')'=xy+x'y'$, which is the required EXNOR expression.



(a) with AND-OR-NOT gates



(b) with NAND gates


x
2
(x+(x+y)')' = x'(x+y) = x'y
(x+y)'
1
4
F = (x'y+xy')' = xy + x'y' = x ⊗ y
3
y
((x+y)'+y)' = (x+y)y' = xy'

(c) with NOR gates

Figure 4.10 Implementation of EXNOR gate with other gates.

4.7 Integrated Circuit Logic Gates

4.7.1 IC Logic Families

Digital circuits are invariably constructed with *integrated circuits (ICs)*. Digital IC gates are classified by the specific logic-circuit family to which they belong. Each logic family has its own basic electronic circuit upon which more complex digital circuits and functions are developed. The basic circuit in each family is either a NAND or a NOR gate. The electronic components employed in the construction of the basic circuit are usually used to name the logic family. Many different logic families of digital ICs have been introduced commercially. The ones that have achieved widespread popularity are listed below:

TTL	Transistor-transistor logic
ECL	Emitter-coupled logic
MOS	Metal-oxide semiconductor
CMOS	Complementary metal-oxide semiconductor
I^2L	Integrated-injection logic

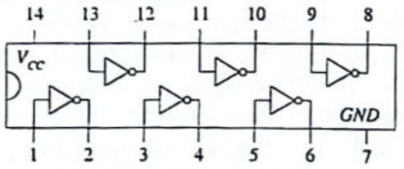
TTL has an extensive list of digital functions and is currently the most popular logic family. ECL is used in systems requiring high-speed operations. MOS and I^2L are used in circuits requiring high component density. CMOS is used in systems requiring low power consumption.

Because of the high density with which transistors can be fabricated in MOS and I^2L, these two families are mostly used for LSI functions. The other three families, TTL, ECL, and CMOS, have LSI devices and also a large number of MSI and SSI devices.

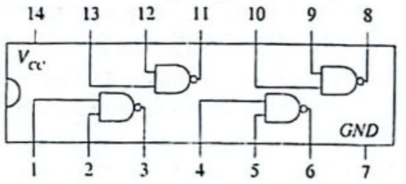
SSI devices are those that come with a small number of gates in one IC package. Some typical TTL SSI circuits are shown in Figure 4.11. Each IC is enclosed with a 14-pin package. The pins are numbered along the two sides of the package and specify the connections that can be made. The gates drawn inside the ICs are for information only and cannot be seen.

TTL ICs are usually distinguished by numerical designation as the 5400 and 7400 series. The former has a wide operating-temperature range, suitable for military use, and the latter has a narrower temperature range, suitable for industrial use. The numeric designation of the 7400 series means that IC package are numbered as 7400, 7401, 7402, etc. Some vendors make available TTL ICs with different numerical designation, such as the 9000 or the 8000 series. Figure 4.11 shows two TTL SSI circuits. The 7404 provides six (hex) NOT gates (inverters) in a package. The 7400 provides four (quadruple) 2-input NAND gates. The terminal marked V_{CC} and *GND* are the power supply pins which require a voltage of 5 volts for proper operation.

The most common ECL type is designated as the 10000 series and the CMOS circuits are designated as the 4000 series.



(a) 7404 - Hex NOT gates (inverters)

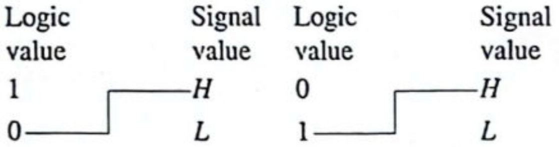


(b) 7400 - Quadruple 2-input NAND gates

Figure 4.11 Some typical TTL SSI gates.

4.7.2 Logic Conventions

The binary signal at the inputs or output of any gate can have one of two values, except during transition. One signal value represents 1 and the other 0. Since two signal values are assigned to two logic values, there exist two different assignments of signals to logic. Because of the principle of duality of Boolean algebra, an interchange of signal-value assignment results in a dual-function implementation. Consider the two values of a binary signal as shown in Figure 4.12. One value must be higher than the other since the two values must be different in order to distinguish between them. We designate the higher level by H and the lower level by L. There are two choices for logic-value assignment. Choosing the high-level H to represent 1, as shown in Figure 4.12(a), defines a *positive-logic* system. Choosing the low-level L to represent 1, as shown in Figure 4.12(b), defines a *negative-logic* system.



(a) Positive logic (b) Negative logic

Figure 4.12 Signal-amplitude assignment and type of logic.

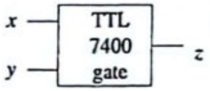
Integrated-circuit data sheets define digital functions not in terms of 1 or 0, but rather in terms of H and L levels. It is up to the user to decide on a positive or negative logic assignment. In TTL, supply voltage $V_{CC} = 5$ volts, high-level voltage range is 2.4 – 5 volts, and low-level voltage range is 0 – 0.4 volts. It has typical values of $H = 3.5$ volts and $L = 0.2$ volts.

In light of this discussion, it would be necessary to justify the logic symbols used for the ICs listed in Figure 4.11. Take, for example, one of the gates of the 7400 IC. This gate is shown in block diagram form in Figure 4.13(a). The manufacturer's truth table for this gate given in a data sheet is shown in Figure 4.13(b). This specifies the physical behavior of the gate, with H being typically 3.5 volts and L being 0.2 volt. This physical gate can function as either a NAND or NOR gate, depending on the polarity assignment.

The truth table of Figure 4.13(c) assumes positive-logic assignment with $H = 1$ and $L = 0$. Checking this truth table with the truth tables in Figure 4.1, we recognize it as a NAND gate. The graphic symbol for a positive-logic NAND gate is shown in Figure 4.13(d) and is similar to the one adopted previously.

Now consider the negative-logic assignment for this physical gate with $L = 1$ and $H = 0$. The result is the truth table shown in Figure 4.13(e). This table can be recognized to represent the NOR function even though its entries are listed backwards. The graphic symbol for a negative-logic NOR gate is shown in Figure 4.13(f). The small triangle in the input and output wires designates a *polarity indicator*. The presence of this polarity indicator along a terminal indicates that negative logic is assigned to the terminal. Thus, the same physical gate can function either as a positive-logic NAND or as a negative-logic NOR. The one drawn in the diagram is completely dependant on the polarity assignment that the designer wishes to employ.

In a similar manner, it is possible to show that a positive-logic NOR is the same physical gate as a negative-logic NAND. The same relation holds between AND and OR gates or between EXOR and EXNOR gates. In any case, if negative logic is assumed in any input or output terminal, it is necessary to include the polarity indicator triangle symbol along the terminal. Note that the ICs presented in Figure 4.11 are shown with their positive-logic graphic symbols.



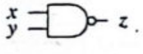
(a) Gate block diagram

x	y	z
L	L	H
L	H	H
H	L	H
H	H	L

(b) Truth table in terms of H and L

x	y	z
0	0	1
0	1	1
1	0	1
1	1	0

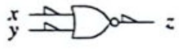
(c) Truth table for positive logic; $H = 1, L = 0$



(d) Graphic symbol for positive logic NAND gate

x	y	z
1	1	0
1	0	0
0	1	0
0	0	1

(e) Truth table for negative logic; $L = 1, H = 0$



(f) Graphic symbol for negative logic NOR gate

Figure 4.13 Demonstration of positive and negative logic.

4.7.3 Characteristics of IC Logic Families

The most important parameters of a IC logic family are *fan-out*, *power dissipation*, *propagation delay*, and *noise margin*.

The *fan-out* is the maximum number of inputs that can be connected to the output of a gate, and it is expressed by a number. Exceeding the fan-out capacity may cause a malfunction of the gate. The fan-out capacity of a gate must be considered when simplifying Boolean functions. Care must be taken not to develop expressions that result in an overloaded gate. Buffers are sometimes employed to provide additional driving capabilities for heavy loads.

Power dissipation represents the actual power dissipated in the gate and is expressed in milliwatts (mW). The total power dissipated in a system is the sum total of the power dissipated in all ICs.

The signals through a gate take a certain amount of time to propagate from the inputs to the output. This interval of time is defined as the *propagation delay* of the gate. Propagation delay is expressed in nanoseconds (ns). The signals that travel from the inputs of a digital circuit to its output pass through a series of gates. The sum of the propagation delays through the gates is the total propagation delay of the circuit. When speed of operation is important, each gate must have a small propagation delay and the digital circuit must have a minimum number of series gates between inputs and outputs.

Noise is a term used to denote an undesirable signal that is superimposed upon the normal operating signal. *Noise margin* represents the maximum noise signal that can be tolerated by the gate and is expressed in volts (V).

There are many versions of TTL, the common are standard TTL, Schottky TTL, and low-power Schottky TTL. The typical characteristics of these versions are listed in Table 4.3.

Table 4.3 Typical characteristics of TTL logic family.

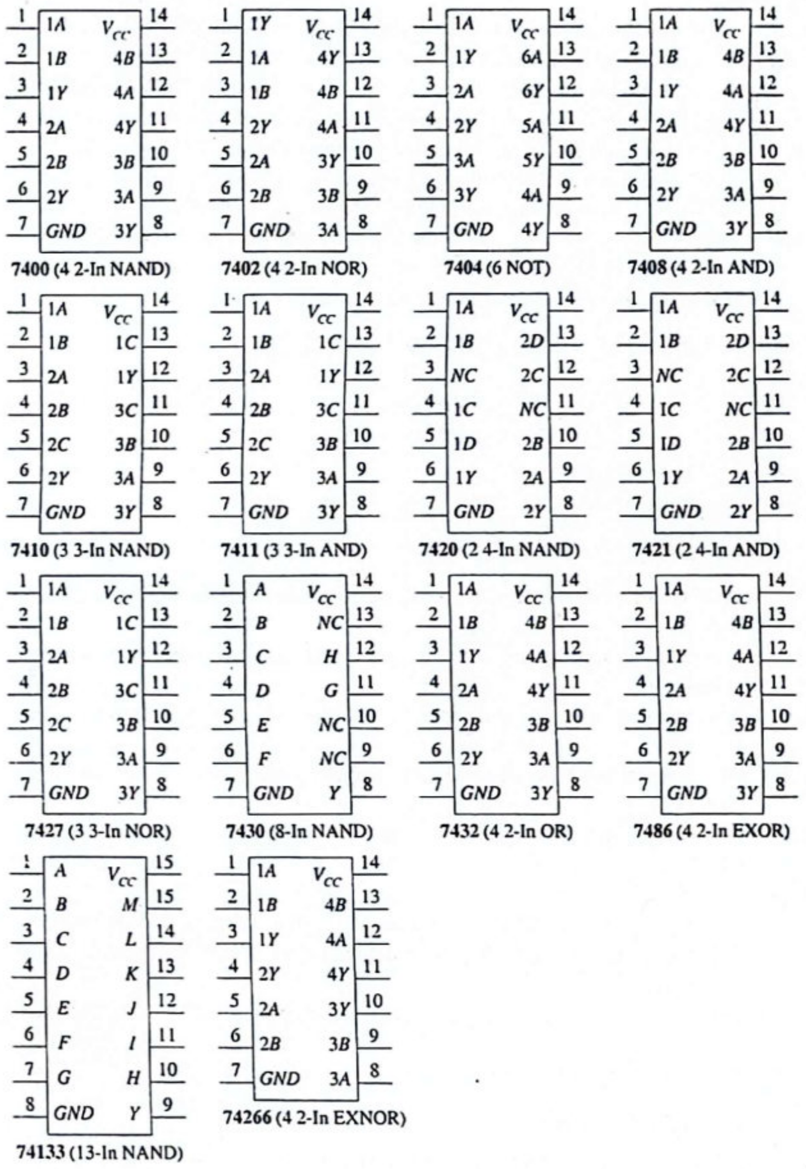
Version of TTL logic family	Fan out	Power dissipation (mW)	Propagation delay (nS)	Noise margin (V)
Standard TTL	10	10	10	0.4
Schottky TTL	10	22	3	0.4
Low-power Schottky TTL	20	2	10	0.4

4.8 Some Available TTL Logic Gates

Some of the available TTL logic gates are listed in Table 4.4. The names of the gates are given for positive-logic convention and their pin connection is shown in Figure 4.14. In Figure 4.14, 1, 2, etc. represent gate number within the IC; *A*, *B*, *C* etc. represent inputs of the corresponding gate; *Y* represents output of the corresponding gate; and *NC* represents "No Connection" pin, that is, the pin is not used. For example, 2*A* represents input *A* of gate 2 and 2*Y* represents output of gate 2.

Table 4.4 Some of the available TTL IC gates.

IC No	Description	IC No	Description
7400	Quad 2-input NAND gates	7421	Dual 4-input AND gates
7402	Quad 2-input NOR gates	7427	Triple 3-input NOR gates
7404	Hex NOT gates (inverters)	7430	8-input NAND gate
7408	Quad 2-input AND gates	7432	Quad 2-input OR gates
7410	Triple 3-input NAND gates	7486	Quad 2-input EXOR gates
7411	Triple 3-input AND gates	74133	13-input NAND gate
7420	Dual 4-input NAND gates	74266	Quad 2-input EXNOR gates


1 1A, 2 1B, 3 1Y, 4 2A, 5 2B, 6 2Y, 7 GND, 8 3Y, 9 3A, 10 3B, 11 4Y, 12 4A, 13 4B, 14 Vcc
1 1Y, 2 1A, 3 1B, 4 2Y, 5 2A, 6 2B, 7 GND, 8 3A, 9 3B, 10 3Y, 11 4A, 12 4B, 13 4Y, 14 Vcc
1 1A, 2 1Y, 3 2A, 4 2Y, 5 3A, 6 3Y, 7 GND, 8 4Y, 9 4A, 10 5Y, 11 5A, 12 6Y, 13 6A, 14 Vcc
1 1A, 2 1B, 3 1Y, 4 2A, 5 2B, 6 2Y, 7 GND, 8 3Y, 9 3A, 10 3B, 11 4Y, 12 4A, 13 4B, 14 Vcc
1 1A, 2 1B, 3 2A, 4 2B, 5 2C, 6 2Y, 7 GND, 8 3Y, 9 3A, 10 3B, 11 3C, 12 1Y, 13 1C, 14 Vcc
1 1A, 2 1B, 3 2A, 4 2B, 5 2C, 6 2Y, 7 GND, 8 3Y, 9 3A, 10 3B, 11 3C, 12 1Y, 13 1C, 14 Vcc
1 1A, 2 1B, 3 NC, 4 1C, 5 1D, 6 1Y, 7 GND, 8 2Y, 9 2A, 10 2B, 11 NC, 12 2C, 13 2D, 14 Vcc
1 1A, 2 1B, 3 NC, 4 1C, 5 1D, 6 1Y, 7 GND, 8 2Y, 9 2A, 10 2B, 11 NC, 12 2C, 13 2D, 14 Vcc
1 1A, 2 1B, 3 2A, 4 2B, 5 2C, 6 2Y, 7 GND, 8 3Y, 9 3A, 10 3B, 11 3C, 12 1Y, 13 1C, 14 Vcc
1 A, 2 B, 3 C, 4 D, 5 E, 6 F, 7 GND, 8 Y, 9 NC, 10 NC, 11 G, 12 H, 13 NC, 14 Vcc
1 1A, 2 1B, 3 1Y, 4 2A, 5 2B, 6 2Y, 7 GND, 8 3Y, 9 3A, 10 3B, 11 4Y, 12 4A, 13 4B, 14 Vcc
1 1A, 2 1B, 3 1Y, 4 2A, 5 2B, 6 2Y, 7 GND, 8 3Y, 9 3A, 10 3B, 11 4Y, 12 4A, 13 4B, 14 Vcc
1 A, 2 B, 3 C, 4 D, 5 E, 6 F, 7 G, 8 GND, 9 Y, 10 H, 11 I, 12 J, 13 K, 14 L, 15 M, 15 Vcc
1 1A, 2 1B, 3 1Y, 4 2Y, 5 2A, 6 2B, 7 GND, 8 3A, 9 3B, 10 3Y, 11 4Y, 12 4A, 13 4B, 14 Vcc

7400 (4 2-In NAND)
7402 (4 2-In NOR)
7404 (6 NOT)
7408 (4 2-In AND)
7410 (3 3-In NAND)
7411 (3 3-In AND)
7420 (2 4-In NAND)
7421 (2 4-In AND)
7427 (3 3-In NOR)
7430 (8-In NAND)
7432 (4 2-In OR)
7486 (4 2-In EXOR)
74133 (13-In NAND)
74266 (4 2-In EXNOR)

Figure 4.14 Pin connection of some TTL logic gates.

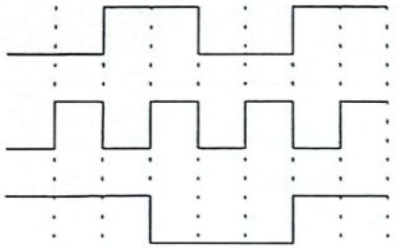
Exercises

4.1 Show that the dual of the EXOR is EXNOR.

4.2 By substituting the Boolean function, show that
- (a) The EXOR and EXNOR operations are commutative and associative.
- (b) The NOR and NAND operators are not distributive.

4.3 A *majority* gate is a digital circuit whose output is equal to 1 if the majority of the inputs are 1s. The output is 0 otherwise. By means of a truth table, find the Boolean function implemented by a 3-input majority gate.

4.4 Draw the output wave form, if the input waveforms are as given below, for (a) 3-input AND gate, (b) 3-input OR gate, (c) 3-input NAND gate, (d) 3-input NOR gate, (e) 3-input EXOR gate, and (f) 3-input EXNOR gate.



4.5 How many different ways do we have to implement the NOT operation in logic circuit?

4.6 How many different ways do we have to implement AND operation in a logic circuit?

4.7 How many different ways do we have to implement OR operation in a logic circuit?

4.8 How many alternate symbols do we have to use for NAND, NOR, and NOT gates?

4.9 How many different ways do we have to implement EXOR and EXNOR gates?

4.10 Show that
- (a) A positive-logic AND gate is a negative-logic OR gate, and vice versa.
- (b) A positive-logic EXOR gate is a negative-logic EXNOR gate, and vice versa.
- (c) A positive-logic NAND gate is a negative-logic NOR gate, and vice versa.

4.11 Given the Boolean function

$$F = xy + x'y' + y'z$$

(a) Implement it with AND, OR, and NOT gates.
(b) Implement it with only OR and NOT gates.
(c) Implement it with only AND and NOT gates.

Further Reading

1. *The TTL Data Book.* Dallas, Texas Instruments, 1988.
2. Tsutomu Sasao (editor), *Logic Synthesis and Optimization*, Kluwer Academic Publishers, 1993.
3. *Microprocessor Data Hand Book*, Revised Ed. BPB Publications, India, 2000.
4. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
5. John P. Uyemura, *Digital Systems Design: An Integrated Approach*, Brooks/Cole Publishing Company, 2000.
6. D. C. Green, *Digital Electronics*, 5th ed., Pearson Education, 2001.
7. Ronald J. Tocci and Neal S. Widmer, *Digital Systems: Principles and Applications*, 8th ed., Prentice-Hall, Inc., 2001.
8. V. K. Jain, *Switching Theory and Digital Electronics*, Khanna Publishers, India, 2002.

CHAPTER 5
SIMPLIFICATION OF BOOLEAN FUNCTIONS

5.1 Introduction

In section 3.4, we introduced the logic diagram of a Boolean function. For implementing a sum of product expression, each product term is implemented by an AND gate. The outputs of the AND gates are then ORed with a multi-input OR gate. Thus, the number of AND gates is equal to the number of product terms in the expression. On the other hand, for implementing a product of sums expression, each sum term is implemented by an OR gate. The outputs of the OR gates are then ANDed with a multi-input AND gate. Thus, the number of OR gates is equal to the number of sum terms in the expression. Moreover, the number of input connections in both the cases is equal to the number of literals in the expression. These facts are shown in Figure 5.1. Note that in the logic diagram of Figure 5.1, it is assumed that all of the input variables are simultaneously available in both their uncomplemented (normal) and complemented form. To show this, NOT gates are used with all the inputs to indicate their complemented output. For this reason, the NOT gates are excluded from determining the cost of the logic circuit. The same convention of showing both the uncomplemented and complemented forms of input variables will be followed throughout the book unless otherwise stated.

The number of AND gates in sum of products implementation and the number of OR gates in product of sums implementation constitute the major cost of the logic circuit implementation. Moreover, an increased number of input connections decreases the reliability of the circuit, because outage of a single input connection will make the circuit malfunctioning. In section 3.4, we also found that algebraic expression of a Boolean function is not unique, that is, it is possible to find more than one algebraic expression that specify the same function. Therefore, it is important to find out the expression with minimum number of terms and with minimum number of literals. Finding such an expression is referred to as *simplification* or *minimization* of Boolean functions.

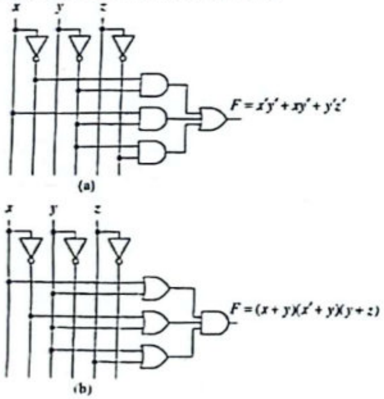


Figure 5.1 Logic circuit implementation of (a) $F = x'y' + xy' + y'z'$ and (b) $F = (x + y)(x' + y)(y + z)$.

In this chapter, we shall discuss four major approaches for simplification of Boolean functions: *algebraic simplification*, *Karnaugh-map method of simplification*, *Quine-McCluskey method of simplification*, and *iterated consensus method of simplification*.

5.2 Algebraic Simplification of Boolean Functions

The Boolean algebra postulates and theorems listed in Table 3.4 can be used to simplify Boolean functions. Unfortunately, it is not always obvious which postulates and theorems should be applied in order to produce the simplest result. Furthermore, there is no easy way to tell whether the simplified expression is in its simplest form or whether it could have been simplified further. Thus, algebraic simplification often becomes a process of trial and error. The quality of algebraic simplification greatly depends on inspiration and experience.

Though, there is no straightforward way of algebraic simplification, the following two steps can be followed:

1. The original expression is put into the sum of products form by using DeMorgan's theorem and distributive law.
2. Once it is in sum of products form, the product terms are checked for common factors, and factoring is performed whenever possible. Sometimes, the expression is expanded by duplicating some product terms to ease factoring. Then postulate and theorems of Boolean algebra are used to eliminate one or more terms.

The above steps are illustrated using the following examples.

Example 1: $F = ABC + AB'(A'C')'$

$= ABC + AB'(A + C)$ by theorem 5 (DeMorgan's theorem)

$= ABC + AB' + AB'C$ by postulate 4 (distributive law)

$= AC(B + B') + AB'$ by postulate 4

$= AC + AB'$ by postulate 5

Example 2: $F = ABC + ABC' + AB'C$

$= ABC + ABC' + ABC + AB'C$ by expanding using theorem 1

$= AB(C + C') + AC(B + B')$ by postulate 4 (distributive law)

$= AB + AC$ by postulate 5

Example 3: $F = A'C(A'BD)' + A'BC'D' + AB'C$

$= A'C(A + B' + D') + A'BC'D' + AB'C$ by theorem 5 (DeMorgan's theorem)

$= A'CA + A'CB' + A'CD' + A'BC'D' + AB'C$ by postulate 4 (Distributive law)

$= A'B'C + A'CD' + A'BC'D' + AB'C$ using postulate 5

$$= B'C(A'+A)+A'CD'+A'BC'D' \qquad \text{using postulate 4}$$

$$= B'C+A'CD'+A'BC'D' \qquad \text{using postulate 5}$$

5.3 Prime Implicants

Karnaugh Map method, Quine-McCluskey method, and iterated consensus method of simplification of Boolean functions will be discussed in the next three sections. Before that concept of prime implicants is worth discussing.

Implication

A Boolean function F_1 implies a Boolean function F_2 (or conversely a Boolean function F_2 is implied by a Boolean function F_1), if $F_2 = 1$ whenever $F_1 = 1$ and F_2 may be either 0 or 1 when $F_1 = 0$. This is written as $F_1 \subseteq F_2$ and read as "F_1 *implies* F_2" or "F_1 *is included in* F_2" or "F_1 *is covered by* F_2" or "F_2 *covers* F_1". F_1 *is said to be the implicant of* F_2. As an example, consider the following functions:

$$F_1(A,B,C) = \sum(0,2,5,7)$$

$$F_2(A,B,C) = \sum(0,5)$$

$$F_3(A,B,C) = \sum(0,1,3,5,7)$$

It can be noticed that $F_2 \subseteq F_1$, $F_2 \subseteq F_3$, but $F_3 \not\subseteq F_1$, $F_1 \not\subseteq F_3$.

From the definition of implication, the following properties may be noted.

(1) If $F_1 \subseteq F_2$, then $F_1 + F_2 = F_2$ and $F_1 \cdot F_2 = F_1$.

(2) If T, T_1, and T_2 are product terms such that $T = T_1 \cdot T_2$, then $T \subseteq T_2$, since $T + T_2 = T_1 \cdot T_2 + T_2 = T_2$.

Prime Implicants

A product term P is a *prime implicant (PI)* of a Boolean function F if

(1) $P \subseteq F$, and

(2) if any literal is deleted from P so as to produce a new product term Q then $Q \not\subseteq F$.

Consider the Boolean function

$$F(A,B,C) = ABC + ABC' + A'BC + AB'C + A'B'C'$$

After simplification it gives

$$F(A,B,C) = AB + BC + AC + A'B'C'$$

Now ABC, ABC', $A'BC$, and $AB'C$ cannot be prime implicants of the function F, since it is possible to delete a literal from them and still the resulting term is included in F. But AB, BC, AC, and $A'B'C'$ are prime implicants of the function, since it satisfies the condition (1) and (2). If any of the literals is deleted from these terms, the resulting term is not included in F.

A Prime implicant P is said to cover a set of minterms $T_1, T_2, \cdots, T_m$ if $T_1 + T_2 + \cdots + T_m \subseteq P$. A Boolean function is covered by a set of product terms (usually prime implicants) if the sum of those product terms (prime implicants) equals the function. A minimal cover is obtained by selecting the minimum number of prime implicants of the function.

The term *essential prime implicant* denotes a prime implicant that covers at least one minterm (not a don't care term) that can not be covered by any other prime implicant. The essential prime implicants, therefore, must be included to get a minimum cover of the function. The prime implicants excluding the essential prime implicants are the *non-essential prime implicants*. After inclusion of the essential prime implicants in the cover of the function, we are left with some minterms to be covered by the set of non-essential prime implicants. All of the non-essential prime implicants are not usually needed for this purpose and a subset of the non-essential prime implicants is sufficient for covering the remaining minterms. These prime implicants are known as *secondary-essential prime implicants*.

Minimization of Boolean Functions

The minimization procedure for Boolean functions consists of the following steps:

(1) The determination of all prime implicants of the function.

(2) Finding out the essential prime implicants.

(3) From the non-essential prime implicants, finding out a minimum set of secondary-essential prime implicants so that together with essential prime implicants they cover the function.

5.4 Karnaugh Map Method for Simplification of Boolean Functions

5.4.1 The Karnaugh Map

The *Karnaugh map* (*K-map* in short) is an special pictorial form of a truth table. This map provides a simple straightforward procedure for representation and simplification of Boolean functions. This map method is first proposed by Veitch and then slightly modified by Karnaugh. Perhaps the K-map is the most extensively used tool for simplification of Boolean functions. Although the technique may be used for any number of variables, it is generally used up to six variables beyond which it becomes very cumbersome. Moreover, K-map method of simplification for five and six variables are not so simple to use, they can best be done by Quine-McCluskey method or iterated consensus method. In this section the structure of two-, three-, and four-variable K-maps are presented.

The K-map is a diagram made up of 2^n *cells* corresponding to 2^n rows of a truth table for n-variable function. The structures of two-, three-, and four-variable K-maps are shown in Figure 5.2. Each cell corresponds to one of the 2^n combinations of n variables and for each row of the truth table there is one specific cell in the K-map. The variables have been designated as A, B, C, and D, and the binary numbers formed by them are taken as AB, ABC, and $ABCD$ for two, three, and four variables respectively. In each map the variables and all possible values of the variables are indicated (the first bit corresponds to the first variable and the second bit corresponds to the second variable) to identify the cells. Gray code sequence has been used for the identification of cells. The reason for using Gray code sequence will be discussed in subsequent discussions. The decimal number corresponding to each cell is written inside the cell in Figure 5.2.

A \ B	0	1
0	0	1
1	2	3

A \ BC	00	01	11	10
0	0	1	3	2
1	4	5	7	6

AB \ CD	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

(a) Two-variable K- (b) Three-variable K-ma(p) Four-variable K-map

Figure 5.2 The Karnaugh map.

Figure 5.3 shows the minterm/maxterm corresponding to each cell and the term is written inside the cell for clear understanding. Notice that for minterm a variable appears primed if the corresponding marking is 0 and unprimed if 1. Also notice that for maxterm a variable appears primed if the corresponding marking is 1 and unprimed if 0.

The columns and rows of the K-map are marked using the gray code sequence. The characteristic of this sequence is that only one bit changes from 1 to 0 or from 0 to 1 in adjacent cells. *Adjacent cells* are defined to be cells horizontally or vertically next to each other. In addition, the map is considered to lie on a surface with the top and bottom edges, as well as the right and left edges, touching each other to form adjacent cells. For example, in a four-variable K-map, cell 0000 and cell 0010 form adjacent cells, as do cell 0011 and cell 1011.

A \ B	0	1
0	$A'B'$	$A'B$
1	AB'	AB

(a)

A \ B	0	1
0	$A+B$	$A+B'$
1	$A'+B$	$A'+B'$

(b)

A \ BC	00	01	11	10
0	$A'B'C'$	$A'B'C$	$A'BC$	$A'BC'$
1	$AB'C'$	$AB'C$	ABC	ABC'

(c)

A \ BC	00	01	11	10
0	$A+B+C$	$A+B+C'$	$A+B'+C'$	$A+B'+C$
1	$A'+B+C$	$A'+B+C'$	$A'+B'+C'$	$A'+B'+C$

(d)

AB \ CD	00	01	11	10
00	$A'B'C'D'$	$A'B'C'D$	$A'B'CD$	$A'B'CD'$
01	$A'BC'D'$	$A'BC'D$	$A'BCD$	$A'BCD'$
11	$ABC'D'$	$ABC'D$	$ABCD$	$ABCD'$
10	$AB'C'D'$	$AB'C'D$	$AB'CD$	$AB'CD'$

(e)

AB \ CD	00	01	11	10
00	$A+B+C+D$	$A+B+C+D'$	$A+B+C'+D'$	$A+B+C'+D$
01	$A+B'+C+D$	$A+B'+C+D'$	$A+B'+C'+D'$	$A+B'+C'+D$
11	$A'+B'+C+D$	$A'+B'+C+D'$	$A'+B'+C'+D'$	$A'+B'+C'+D$
10	$A'+B+C+D$	$A'+B+C+D'$	$A'+B+C'+D'$	$A'+B+C'+D$

(f)

Figure 5.3 Minterm/maxterm corresponding to each cell of K-maps.

5.4.2 Representation of Truth Table on a K-map

For representing a truth table on a K-map, the value of the function (0, 1, or x) for each row of the truth table is entered inside the cell corresponding to that row. Figure 5.4 shows a truth table for a three-variable function and its representation on a three-variable K-map. The cell on the K-map corresponding to a row of the truth table is identified by concatenating the row marking and the column marking of the K-map and then comparing that value with the row of the truth table. For example, in Figure 5.4(b), the second row marking is 1 and the third column marking is 11; their concatenation results in 111. Therefore, the cell common to the second row (1) and the third column (11) corresponds to the row 111 of the truth table. In the example of Figure 5.4, value of the function for the row 111 in the truth table is 1 and a 1 is entered in the identified cell. Alternatively, the cell can be identified by using the equivalent decimal number of the row of the truth table and comparing it with the cell number as shown in Figure 5.2. For example, the decimal equivalent of the row 100 is 4 and the cell 4 is common to the second row and the first column of a three-variable K-map. In the example of Figure 5.4, a x is entered in that cell.

A	B	C	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	x
1	0	1	0
1	1	0	x
1	1	1	1

A \ BC	00	01	11	10
0	0	1	0	1
1	x	0	1	x

(a) Truth Table (b) K-map representing the truth table

Figure 5.4 Representation of truth table on a K-map.

5.4.3 Representation of Sum of Products Function on a K-map

Representation of Canonical Sum of Products Function on a K-map

A canonical sum of products function can be represented on a K-map by simply entering 1s in the cell of the K-map corresponding to each minterm present in the function. For example, the function

$$F = A'B'C + A'BC' + AB'C' + ABC$$

is represented on the K-map of Figure 5.5.

A \ BC	00	01	11	10
0		1		1
1	1		1	

Figure 5.5 Representation of the function
$F = A'B'C + A'BC' + AB'C' + ABC$ on a K-map.

Representation of Noncanonical Sum of Products Function on a K-map

In noncanonical sum of products function, some of the product terms have less than n literals, where n is the number of the variables in the function. These terms are represented on the K-map by more than one cell. For example, the function

$$F = A'C + A'B + AB'C + BC$$

is represented on the K-map of Figure 5.6. In Figure 5.6, to find the cells corresponding to $A'C$, we form the coincidence of A' (first row) and C (two middle columns) and obtain cells 001 and 011. 1s are entered in these cells. Note that when entering 1s in the cells, it is possible to find a 1 already placed there by a preceding term. The second term $A'B$ has 1s in cells 011 and 010, but cell 011 is common to the first term $A'C$ and only one 1 is marked in the cell 010 for the term $A'B$. The function has five minterms as indicated by the five cells marked with 1s. Note that the K-map is a useful tool for converting a noncanonical sum of products function into canonical sum of products function.

BC	B'		B	
A	00	01	11	10
A' → 0		1	1	1
A → 1		1	1	
	C'	C		C'

Figure 5.6 Representation of the function $F = A'C + A'B + AB'C + BC$ on a K-map.

5.4.4 Representation of Product of Sums Function on a K-map

Representation of Canonical Product of Sums Function on a K-map

A canonical product of sums function can be represented on a K-map by entering 0s in the cells of K-map corresponding to each maxterm present in the function. For example, the function

$$F = (A + B + C)(A + B' + C')(A' + B + C')(A' + B' + C)$$

is represented on the K-map of Figure 5.7.

A \ BC	00	01	11	10
0	0		0	
1		0		0

Figure 5.7 Representation of the function
$F = (A + B + C)(A + B' + C')(A' + B + C')(A' + B' + C)$ on a K-map.

Representation of Noncanonical Product of Sums Function on a K-map

In noncanonical product of sums function, some of the sum terms have less than *n* literals, where *n* is the number of the variables in the function. These terms are represented on the K-map by more than one cell. For example, the function

$$F=(B+C)(A'+B)(A+B'+C')$$

is represented in Figure 5.8. In Figure 5.8, to find the cells corresponding to $(B+C)$, we form the coincidence of *BC* (first column) and obtain cells 000 and 100. 0s are entered in these cells. The second term $(A'+B)$ has 0s in cells 100 and 101, but cell 100 is common to the first term $(B+C)$ and only one 0 is entered in cell 101 for the term $(A'+B)$. The function has four maxterms as indicated by the four cells marked with 0s. Also note that the K-map is a useful tool for converting noncanonical product of sums function into canonical product of sums function.

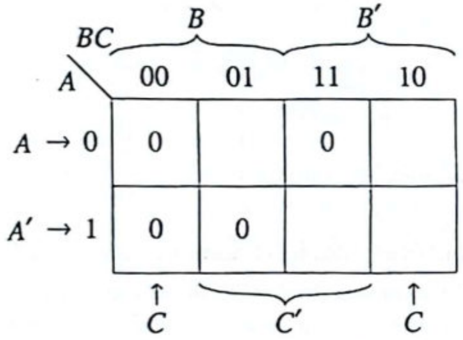


Figure 5.8 Representation of the function $F=(B+C)(A'+B)(A+B'+C')$ on a K-map.

5.4.5 Grouping of Adjacent 1s on a K-map

The property that *two horizontally or vertically adjacent cells differ in only one variable* provides a means of grouping adjacent 1s and simplifying the terms represented by the 1s of that group.

Group of One 1

A single 1 constitutes a group of one 1. A group of one 1 represents a minterm, that is, a product term containing *n* literals, where *n* is the number of variables in the function. A group of one 1 is shown in Figure 5.9(a).

Group of Two 1s

Any two adjacent 1s constitute a group of two 1s. In a group of two 1s, the two minterms can be combined to produce a single product term containing $n-1$ literals. The two minterms differ in only one variable, that is, they are of the form Ax and Ax', where *A* is a product term independent of *x*. These two terms can be combined to produce the product term *A* as follows:

$$Ax+Ax'=A(x+x')=A\cdot 1=A.$$

A group of two 1s is shown in Figure 5.9(a) and their combining is as follows:

$$A'B'CD'+A'BCD'=A'CD'(B'+B)=A'CD'$$

Group of Four 1s

Any four adjacent 1s, which form a square or a rectangle, constitute a group of four 1s. In a group of four 1s, the four minterms can be combined to produce a single product term containing $n-2$ literals. The four minterms differ in two variables, that is, they are of the form $Ax'y'$, $Ax'y$, Axy', and Axy, where *A* is a product term independent of *x* and *y*. These four minterms can be combined to produce the product term *A* as follows:

$$Ax'y'+Ax'y+Axy'+Axy=Ax'(y'+y)+Ax(y'+y)=Ax'+Ax$$
$$=A(x'+x)=A$$

A group of four 1s is shown in Figure 5.9(a) and their combining is as follows:

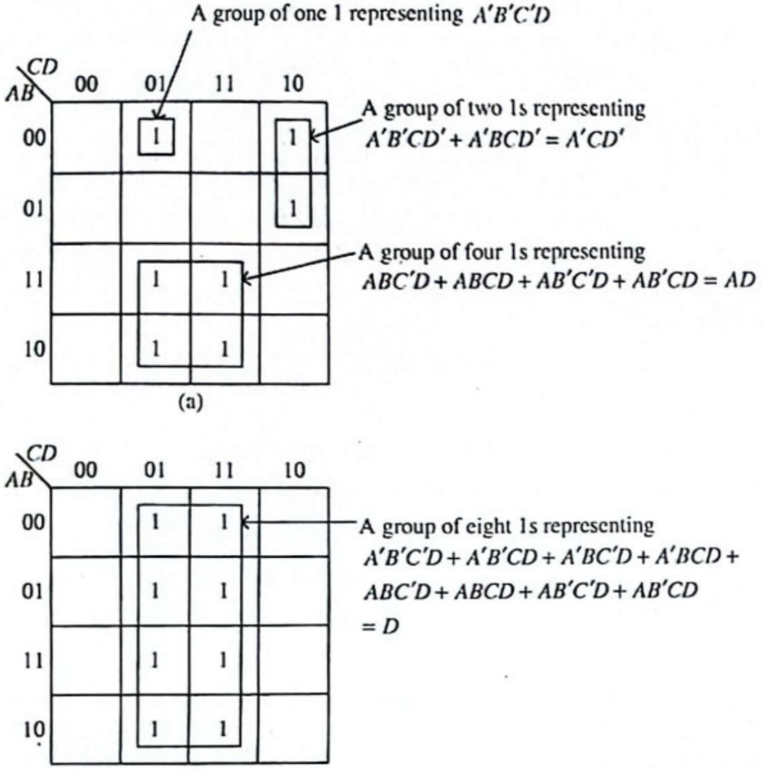
$$ABC'D+ABCD+AB'C'D+AB'CD=ABD(C'+C)+AB'D(C'+C)$$
$$=ABD+AB'D$$
$$=AD(B+B')$$
$$=AD$$

Group of Eight 1s

Any eight adjacent 1s, which form a rectangle, constitute a group of eight 1s. In a group of eight 1s, the eight minterms differ in three variables and they can be combined to produce a single product term containing $n-3$ literals. A group of eight 1s is shown in Figure 5.9 (b) and their combining is as follows:

$$A'B'C'D+A'B'CD+A'BC'D+A'BCD+ABC'D+ABCD+AB'C'D+AB'CD$$
$$=A'B'D(C'+C)+A'BD(C'+C)+ABD(C'+C)+AB'D(C'+C)$$
$$=A'B'D+A'BD+ABD+AB'D$$
$$=A'D(B'+B)+AD(B'+B)$$
$$=A'D+AD$$
$$=D(A'+A)$$
$$=D$$

In general, in an *n*-variable K-map any 2^m adjacent 1s, which form a square or a rectangle, constitute a group of 2^m 1s. The 2^m minterms of the group differ in *m* variables and they can be combined to produce a single product term containing $n-m$ literals. A group of 2^n 1s combines 2^n minterms to produce a constant 1.



A group of one 1 representing $A'B'C'D$

A group of two 1s representing $A'B'CD' + A'BCD' = A'CD'$

A group of four 1s representing $ABC'D + ABCD + AB'C'D + AB'CD = AD$

A group of eight 1s representing $A'B'C'D + A'B'CD + A'BC'D + A'BCD + ABC'D + ABCD + AB'C'D + AB'CD = D$

Figure 5.9 Grouping of adjacent 1s on a K-map for simplification.

Reading Product Term from a Group of 1s

The product term specified by a group of 1s can be read directly from the group. The column and row markings of the group are inspected for any variation in bit positions. The variable corresponding to a varying bit position does not appear in the product term. The literals corresponding to the non-varying bit positions are taken for constructing the product term. Note that a variable appears uncomplemented in the product term if its value is 1 and complemented if 0. For example, consider the group of four 1s in Figure 5.9(a). The column markings of this group are 01 and 11. Here the variable C varies and will not appear in the resulting product term. Similarly, the row markings of the group are 11 and 10 and the varying variable B will not appear in the product term. Thus, the resulting product term will be AD.

Don't Care Terms in a Group of 1s

The output for a don't care term is unspecified. A cell containing a don't care term can be assumed to contain a 1 and can be included in a group. Such inclusion will minimize the function though the operation of the corresponding digital circuit will not be affected, since the don't care term will never appear in the input of the circuit. Inclusion of a don't care term in a group is shown in Figure 5.10. It should be noted that a group of all don't care terms are not considered, that is, in a group there must be at least one 1. This is shown in Figure 5.10.

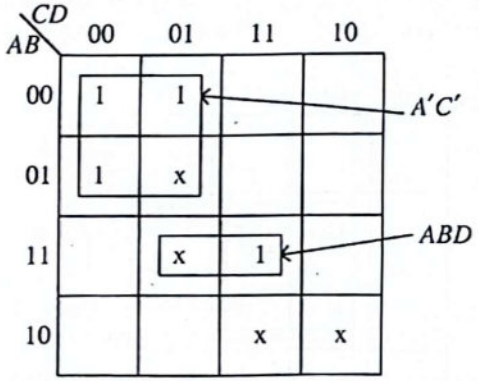


Figure 5.10 Don't care terms in a group of 1s.

Overlapping Groups

A 1 or x can be included in more than one groups for facilitating simplification of the function, since $A + A + \cdots + A + A$. Overlapping groups are shown in Figure 5.11.

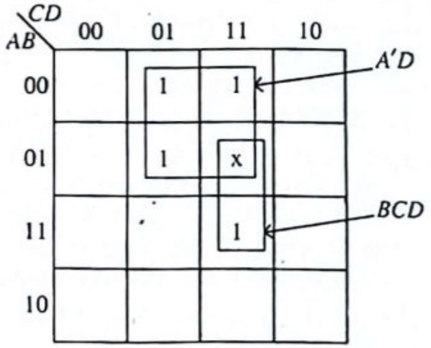


Figure 5.11 Overlapping groups on a K-map.

Prime Implicants on a K-map

A largest possible group of 1s (including any x) constitutes a prime implicant of the function. For example, consider the function represented by the K-map of Figure 5.12. The product term for group 1 is $A'BD$, for group 2 is ABD, and for group 3 is BD. The product terms $A'BD$ and ABD are not prime implicants of the function, since it is possible to delete a literal from them and still the resulting term is included in the function. But BD is a prime implicant of the function. Therefore, a group completely included within another group does not represent a prime implicant of the function. A group representing a prime implicant is an *essential prime implicant* if it covers at least one 1 not being covered by any other group.

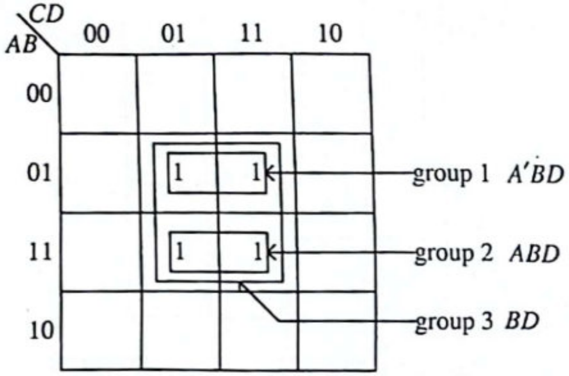


Figure 5.12 Prime Implicants on a K-map.

5.4.6 Grouping of Adjacent 0s on a K-map

In a K-map representing a product of sums function, the grouping of adjacent 0s is similar to the grouping of 1s. A group of one 0 represents a maxterm, that is, a sum term containing n literals, where n is the number of variables in the function. In a group of two 0s, the two maxterms differ in one variable and they are of the form $(A+x)$ and $(A+x')$, where A is a sum term independent of x. They can be combined to produce a single sum term A, containing $n-1$ literals, as follows:

$$(A+x)(A+x') = A\cdot A + A\cdot x' + A\cdot x + x\cdot x'$$
$$= A + A(x'+x) = A + A = A$$

In general, in an n-variable K-map any 2^m adjacent 0s constitute a group of 2^m 0s; the 2^m maxterms of the group differ in m variables and they can be combined to produce a single sum term containing $n-m$ literals. A group of 2^n 0s combines 2^n maxterms to produce a constant 0.

The sum term specified by a group of 0s can be read directly from the K-map in a similar manner as reading a product term specified by a group of 1s. The variable corresponding to a varying bit position does not appear in the sum term. The literals corresponding to the non-varying bit positions are taken for constructing the sum term. Note that a variable appears uncomplemented in the sum term if its value is 0 and complemented if 1.

A cell containing a don't care term can be assumed to contain a 0 and can be included in a group of 0s in a similar as it is done in case of group of 1s. Overlapping groups are also considered in a similar manner as those are considered for groups of 1s. Groups of adjacent 0s are shown in Figure 5.13.

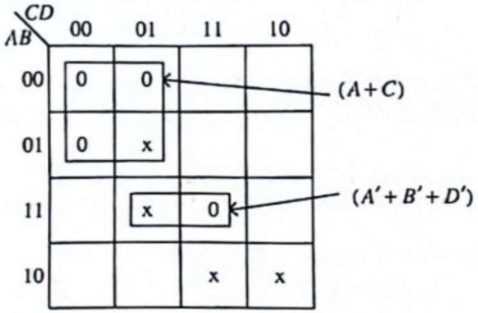


Figure 5.13 Grouping of adjacent 0s on a K-map for simplification.

5.4.7 Simplification of Sum of Products Function using K-map

Simplification Procedure

The simplification of sum of products function using K-map consists of the following steps:

1. Represent the function on a K-map and find all possible groups representing prime implicants of the function.
2. Find all essential prime implicants.
3. Determine the 1s not covered by essential prime implicants. Choose a minimum number of non-essential prime implicants, which cover the remaining 1s. The non-essential prime implicants covering the remaining 1s are referred to as secondary-essential prime implicants.
4. Determine the simplified sum of products function by combining the essential and the secondary-essential prime implicants.

The above procedure is illustrated by the following examples.

Example 1: $F(A,B,C,D) = \sum(0,2,5,7,8,10,15)$

1. The function is represented on the K-map of Figure 5.14. Groups 1, 2, and 3 represent all possible prime implicants of the function. The product term for group 1 is $B'D'$, for group 2 is $A'BD$, and for group 3 is BCD.

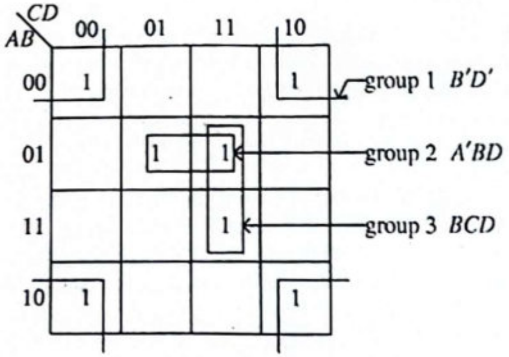


Figure 5.14 Simplification of sum of products function $F(A, B, C, D) = \sum(0,2,5,7,8,10,15)$ using K-map.

2. Groups 1, 2, and 3 are all essential prime implicants.
3. As all the prime implicants are essential prime implicants, all 1s are covered by them.
4. The simplified sum of products function is

$$F(A, B, C, D) = B'D' + A'BD + BCD.$$

Example 2: $F(A, B, C, D) = \sum(1,4,5,6,7,9,10,11,13,14)$

1. The function is represented on the K-map of Figure 5.15. Groups 1, 2, 3, 4, 5, and 6 represent all possible prime implicants of the function. The product term for group 1 is $C'D$, for group 2 is $A'B$, for group 3 is BCD', for group 4 is ACD', for group 5 is $AB'C$, and for group 6 is $AB'D$.

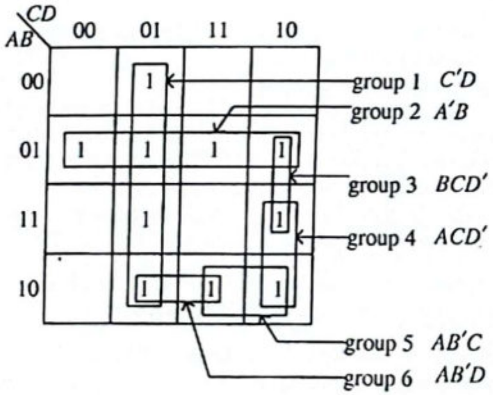


Figure 5.15 Simplification of sum of products function $F(A, B, C, D) = \sum(1,4,5,6,7,9,10,11,13,14)$ using K-map.

2. Groups 1 and 2 are essential prime implicants.
3. Groups 3, 4, 5, and 6 are non-essential prime implicants. The 1s of the cells 1110, 1011, and 1010 are remained to be covered by a minimum subset of the non-essential prime implicants. The selection of the secondary-essential prime implicants can be done in three ways
   - (i) Taking groups 3 and 5.
   - (ii) Taking groups 4 and 6.
   - (iii) Taking groups 4 and 5.
4. The simplified sum of products function is

$$F(A, B, C, D) = C'D + A'B + BCD' + AB'C$$

or

$$F(A, B, C, D) = C'D + A'B + ACD' + AB'D$$

or

$$F(A, B, C, D) = C'D + A'B + ACD' + AB'C.$$

Example 3: $F(A,B,C,D) = \sum(0,1,3,7,8,12) + \sum_{d.c.}(5,10,13,14)$

1. The function is represented on the K-map of Figure 5.16. Groups 1, 2, 3, 4, and 5 represent all possible prime implicants of the function. The product term for group 1 is $A'D$, for group 2 is AD', for group 3 is $A'B'C'$, for group 4 is $B'C'D'$, and for group 5 is ABC'.

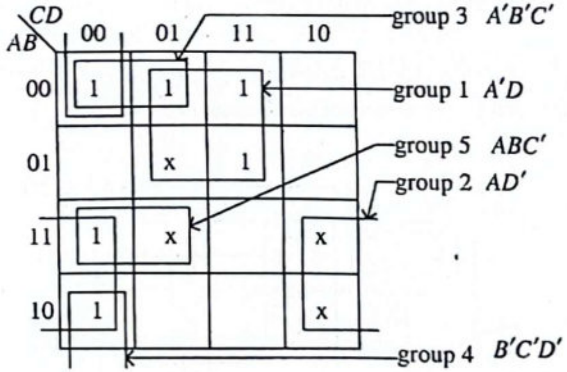


Figure 5.16 Simplification of sum of products function $F(A,B,C,D) = \sum(0,1,3,7,8,12) + \sum_{d.c.}(5,10,13,14)$ using K-map.

2. Group 1 is essential prime implicant.
3. Groups 2, 3, 4, and 5 are non-essential prime implicants. The 1s of the cells 0000, 1100, and 1000 are left to be covered. The 1s of the cells 1100 and 1000 can be covered by choosing group 2. The 1 of the cell 0000 can be covered by choosing either group 3 or 4.

4. The simplified sum of products function is

$$F(A,B,C,D) = A'D + AD' + A'B'C'$$

or

$$F(A,B,C,D) = A'D + AD' + B'C'D'$$

Straightforward Method of Simplification

The simplification procedure discussed above finds all possible simplified forms of the sum of products function. However, in practice, finding a single simplified sum of products function is sufficient. In this case, using intuition and experience, we try to directly identify groups representing essential and secondary-essential prime implicants only rather than identifying all possible groups representing all prime implicants. For this purpose; the following steps can be followed:

1. Represent the function on a K-map.
2. Identify the largest possible group, including don't care terms if possible, such that there is at least one 1 in the group.
3. Identify the next largest possible group, including don't care terms and already covered 1s if possible, such that there is at least one 1 not being covered by previous groups.
4. Repeat step 2 until all 1s are covered.
5. Discard a group if it contains no 1 not being covered by other group.
6. Determine the simplified sum of products function using the product terms of the selected groups.

The above procedure is discussed with the help of the following examples.

Example 1: $F(A,B,C,D) = \sum(0,1,3,7,8,12) + \sum_{d.c.}(5,10,13,14)$

1. The function is represented on the K-map of Figure 5.17.

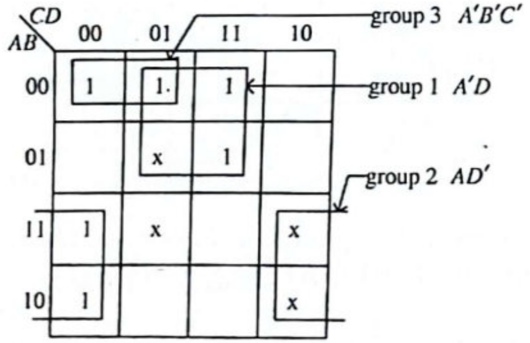


Figure 5.17 Simplification of sum of products function $F(A,B,C,D)=\sum(0,1,3,7,8,12)+\sum_{d.c.}(5,10,13,14)$ using K-map.

2. First identify group 1, which contains three 1s and a don't care term. The product term for group 1 is $A'D$.
3. Then identify group 2, which contains two 1s and two don't care terms. The product term for group 2 is AD'.
4. Then identify group 3, which contains two 1s, among them one 1 is not covered by previous groups. The product term for group 3 is $A'B'C'$.
5. All groups contain at least one 1 not being covered by any other group.
6. The simplified sum of products function is

$$F(A,B,C,D) = A'D + AD' + A'B'C'.$$

Example 2: $F(A,B,C,D)=\sum(1,5,6,7,11,12,13,15)$

1. The function is represented on the K-map of Figure 5.18.
2. First identify group 1, which contains four 1s. The product term for group 1 is BD.
3. Then identify group 2, which contains two 1s, among them one 1 is not covered by previous group. The product term for group 2 is $A'BC$.
4. Similarly identify groups 3, 4, and 5, which contains two 1s each, among them one 1 is not covered by previous groups. The product term for group 3 is ACD, for group 4 is ABC', and for group 5 is $A'C'D$.
5. Group 1 contains four 1s, which are also contained in other groups. So, group 1 is discarded.
6. The simplified sum of products function is

$$F(A,B,C,D) = A'BC + ACD + ABC' + A'C'D.$$

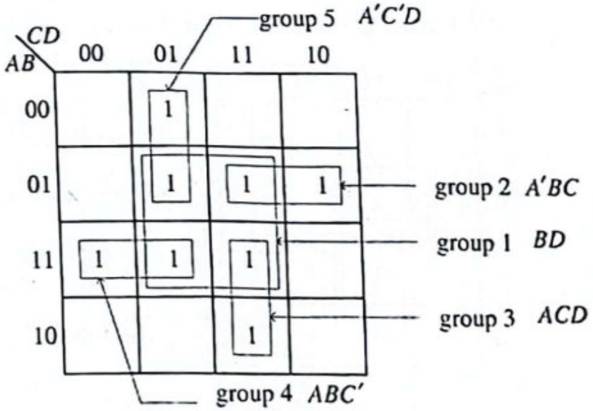


Figure 5.18 Simplification of sum of products function $F(A,B,C,D)=\sum(1,5,6,7,11,12,13,15)$ using K-map.

5.4.8 Simplification of Product of Sums Function using K-map

Simplification Procedure

The simplification procedure for product of sums function is similar to that for sum of products function. The simplification of product of sums function using K-map consists of the following steps:

1. Represent the function on a K-map and find all possible groups.
2. Find all essential groups.
3. Determine the 0s not covered by essential groups. Choose a minimum number of non-essential groups, which cover the remaining 0s. The non-essential groups covering the remaining 0s are referred to as secondary-essential groups.
4. Determine the simplified product of sums function by combining the essential and the secondary-essential groups.

The above procedure is illustrated by the following example.

Example: $F(A,B,C,D)=\prod(2,4,6,9,11,15)\cdot\prod_{d.c.}(5,10,13,14)$.

1. The function is represented on the K-map of Figure 5.19. Groups 1, 2, 3, 4, and 5 are the possible groups. The sum term for group 1 is $(C'+D)$, for group 2 is $(A'+D')$, for group 3 is $(A'+C')$, for group 4 is $(A+B'+D)$ and for group 5 is $(A+B'+C)$.
2. The groups 1 and 2 are essential groups.
3. Groups 3, 4, and 5 are non-essential groups. The cell 0100 containing 0 is left to be covered and can be done by choosing either group 4 or 5.
4. The simplified product of sums function is

$$F(A,B,C,D)=(C'+D)(A'+D')(A+B'+D)$$

or

$$F(A,B,C,D)=(C'+D)(A'+D')(A+B'+C)$$

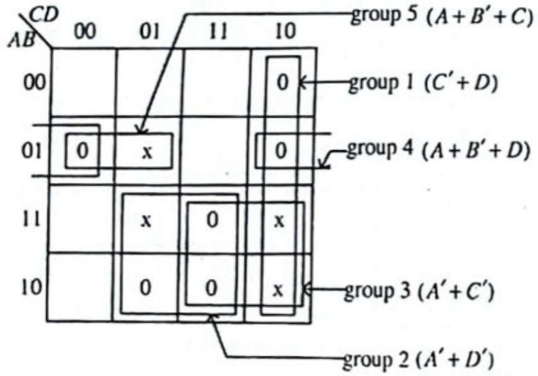


Figure 5.19 Simplification of
$F(A,B,C,D)=\prod(2,4,6,9,11,15)\cdot\prod_{d.c.}(5,10,13,14)$ using K-map.

Straightforward Method of Simplification

The straightforward method of finding a single simplified product of sums function is also similar to that of sum of products function. For this purpose, the following steps can be followed:

1. Represent the function on a K-map.
2. Identify the largest possible group, including don't care terms if possible, such that there is at least one 0 in the group.
3. Identify the next largest possible group, including don't care terms and already covered 0s if possible, such that there is at least one 0 not being covered by previous groups.
4. Repeat step 2 until all 0s are covered.
5. Discard a group if it contains no 0 not being covered by other group.
6. Determine the simplified product of sums function using the sum terms of the selected groups.

The above procedure is discussed with the help of the following examples.

Example: $F(A,B,C,D)=\prod(2,4,6,9,11,15)\cdot\prod_{d.c.}(5,10,13,14)$

1. The function is represented on the K-map of Figure 5.20.
2. First identify group 1, which contains two 0s and two don't care terms. The sum term for group 1 is $(C+D')$.
3. Then identify group 2, which contains three 0s and a don't care term. The sum term for group 2 is $(A'+D')$.
4. Then identify group 3, which contains two 0s, among them one 0 is not covered by previous groups. The sum term for group 3 is $(A+B'+D)$.
5. All groups contain at least one 0 not being covered by any other group.
6. The simplified sum of products function is

$$F(A,B,C,D)=(C+D')(A'+D')(A+B'+D).$$

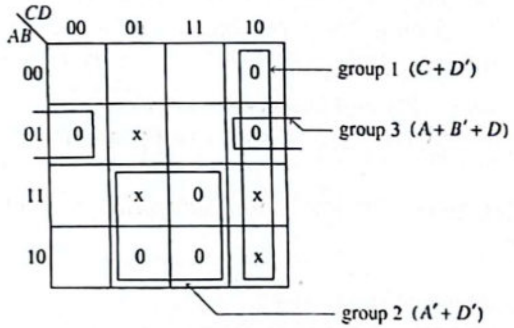


Figure 5.20 Simplification of
$F(A,B,C,D)=\prod(2,4,6,9,11,15)\cdot\prod_{d.c.}(5,10,13,14)$ using K-map.

5.4.7 Simplification of Sum of Products vs. Product of Sums Function

Simplification of Boolean function

$$F(A,B,C,D)=\sum(0,1,2,3,5,7,8,9,11,14)=\prod(4,6,10,12,13,15)$$

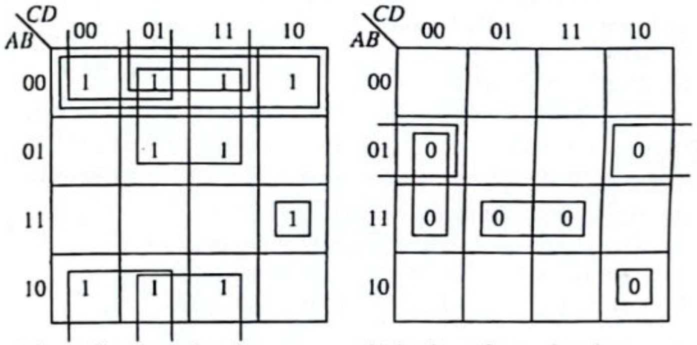
is shown in Figure 5.21. Simplification of sum of products function is shown in Figure 5.21(a), which yields

$$F(A,B,C,D)=A'B'+A'D+B'C'+B'D+ABCD'$$

Simplification of the product of sums function is shown in Figure 5.21(b), which yields

$$F(A,B,C,D)=(B'+C+D)(A'+B'+D')(A+B'+D)(A'+B+C'+D)$$

We observe that the numbers of terms are not same in the two simplifications. In fact, in general the two simplifications will not have the same number of terms and will require different quantities of gates. Therefore, one can obtain both simplifications and select the one that requires minimum gates.



(a) Sum of products function

$F(A,B,C,D)=A'B'+A'D+B'C'+B'D+ABCD'$

(b) Product of sums function

$F(A,B,C,D)=(B'+C+D)(A'+B'+D')(A+B'+D)(A'+B+C'+D)$

Figure 5.21 Simplification of

$$F(A,B,C,D)=\sum(0,1,2,3,5,7,8,9,11,14)=\prod(4,6,10,12,13,15)$$

5.5 Quine-McCluskey Method for Simplification of Boolean Function

5.5.1 The Quine-McCluskey Method

The K-map method is a very useful tool for the simplification of Boolean functions but suffers from some drawbacks. Firstly it depends upon human intuitiveness in forming groups of cells containing 1 or 0. Also the method cannot be applied for the functions having more than six variables and working with five and six variable functions is very cumbersome. The *Quine-McCluskey method*, also known as *tabulation method*, is applicable to functions of any number of variables. It is a specific step-by-step procedure that is guaranteed to produce a simplified sum of products expression for a function. The procedure is also suitable for programming on a digital computer for systematic simplification of large Boolean functions. The method was first formulated by Quine and later improved by McCluskey.

The method consists of the following steps:

1. A set of all prime implicants of the function is obtained.
2. Essential prime implicants and a minimum number of secondary-essential prime implicants are selected so that they cover the function.

5.5.2 Determination of all Prime Implicants

The principle involved in the determination of all prime implicants is the repeated applications of the fact that

$$Ax+Ax'=A(x+x')=A$$

where A is a product term independent of x, on all adjacent pair of product terms so that they generate the set of all prime implicants.

Determination of all Prime Implicants using Binary Notations of Minterms

The procedure for determination of all prime implicants for the function is as follows:

1. Convert the Boolean function into sum of minterms form, if it is not.
2. Express each minterm by its binary representation. Arrange the minterms according to increasing index, where index is defined as the number of 1s in a minterm. Separate the sets of minterms of equal index by lines.
3. Compare each term of index n with each term of index $(n+1)$. For each pair of terms that can combine, write the newly formed terms. Only those terms can be combined that differ only at one bit position. A dash is placed at that bit position in the newly formed term that has been eliminated as a result of combination. The terms that are combined are checked. After all pairs of terms with indices n and $(n+1)$ have been considered, a line is drawn under the last term. The above procedure is repeated for all groups of terms. This completes one stage of elimination.
4. The next stage of elimination is done by using step 3. In this case two terms can be combined only when they have dashes in the same positions. Checks are placed as before. If same term is generated more than once, only one is listed.
5. Repeat step 4 until no new list can be formed.
6. All terms that remain unchecked are prime implicants.

The above procedure is explained for the Boolean function

$$F(A,B,C,D,E)=\sum(0,2,4,6,7,8,10,11,12,13,14,16,18,19,29,30).$$

Step 1: The function is already in sum of minterms form.

Step 2: The result of step 2 is shown in the columns labeled "Index" and "Minterms" of Table 5.1.

Step 3: The result of step 3 is shown in the column labeled "Stage I" of Table 5.1.

Step 4 and 5: The result of step 4 and 5 is shown in the columns labeled "Stage II" and "Stage III" of Table 5.1.

Table 5.1 Determination of all prime implicants of function

$$F(A,B,C,D,E) = \sum(0,2,4,6,7,8,10,11,12,13,14,16,18,19,29,30)$$

Index	Minterms $ABCDE$	Stage I	Stage II	Stage III
0	00000√	000-0√ 00-00√ 0-000√ -0000√	00--0√ 0-0-0√ -00-0 *b* 0--00√	0---0 *a*
1	00010√ 00100√ 01000√ 10000√	00-10√ 0-010√ -0010√ 001-0√ 0-100√ 010-0√ 01-00√ 100-0√	0--10√ 0-1-0√ 01--0√	
2	00110√ 01010√ 01100√ 10010√	0011- *h* 0-110√ 0101- *g* 01-10√ 0110- *f* 011-0√ 1001- *e*		
3	00111√ 01011√ 01101√ 01110√ 10011√	-1101 *d* -1110 *c*		
4	11101√ 11110√			

Step 6: All the unchecked terms are the prime implicants of the function. These are marked from *a* to *h* in Table 5.1. Note that the variable corresponding to a dash is missing from the product terms representing prime implicants. The prime implicants are listed below:

a	0---0	$A'E'$
b	-00-0	$B'C'E'$
c	-1110	$BCDE'$
d	-1101	$BCD'E$
e	1001-	$AB'C'D$
f	0110-	$A'BCD'$
g	0101-	$A'BC'D$
h	0011-	$A'B'CD$

Determination of all Prime Implicants using Decimal Notations of Minterms
The above procedure shown in Table 5.1 consists of comparing terms and finding out terms that differ only at one bit position. This is quite cumbersome and error is likely to occur. The work is considerably reduced if decimal notation is used. Only decimal numbers of the minterms are written and the minterms are grouped according to the number of 1s in their binary representation as earlier. Now when the terms from the adjacent groups are combined, it should be remembered that the number of minterms of the next adjacent group that can combine must be larger by a power of 2, i.e., 1, 2, 4, 8, 16, 32, 64, ..., etc. Thus 2 can be combined with 3, 4, 6, 10, 18, 34, 66, ... , etc. It is quite simple to prove this fact because terms that can be combined must differ only at a single bit position, where the differing bit in the first number is 0 and that in the second number is 1. In binary numbers weight of one bit is equal to 2^i where $i = 0,1,2,3,4,5,6,\cdots$. Therefore, in decimal number the second number will be larger by a power of 2.

Same example is repeated using decimal number in Table 5.2. The number by which two combining terms differ is placed in parenthesis. For example, when 8 and 10 combine, we write 8, 10 (2). Next combination is possible if the numbers in the parenthesis agree in two adjacent groups. The unchecked entries are prime implicants of the function and are marked from *a* to *h*.

Table 5.2 Determination of all prime implicants of function

$F(A,B,C,D,E) = \sum(0,2,4,6,7,8,10,11,12,13,14,16,18,19,29,30)$ using decimal numbers.

Index	Minterms	Stage I	Stage II	Stage III
0	0 √	0, 2 (2) √ 0, 4 (4) √ 0, 8 (8) √ 0, 16 (16) √	0, 2, 4, 6 (2, 4) √ 0, 2, 8, 10 (2, 8) √ 0, 2, 16, 18 (2, 16) *b* 0, 4, 8, 12 (4, 8) √	0, 2, 4, 6, 8, 10, 12, 14 (2, 4, 8) *a*
1	2 √ 4 √ 8 √ 16 √	2, 6(4) √ 2, 10 (8) √ 2, 18 (16) √ 4, 6 (2) √ 4, 12 (8) √ 8, 10 (2) √ 8, 12 (4) √ 16, 18 (2) √	2, 6, 10, 14 (4, 8) √ 4, 6, 12, 14 (2, 8) √ 8, 10, 12, 14 (2, 4)√	
2	6 √ 10 √ 12 √ 18 √	6, 7 (1) *h* 6, 14 (8) √ 10, 11 (1) *g* 10, 14 (4) √ 12, 13 (1) *f* 12, 14 (2) √ 18, 19 (1) *e*		
3	7 √ 11 √ 13 √ 14 √ 19 √	13 ,29 (16) *d* 14, 30 (16) *c*		
4	29 √ 30 √			

An easier method for writing the product terms of the prime implicants is to write down the binary equivalent of any minterm in the prime implicant and then put a dash in the positions whose binary weight corresponds to the numbers given in the parenthesis of the prime implicants. The prime implicants are listed below.

		$2^4 2^3 2^2 2^1 2^0$ 16 8 4 2 1	
a	0, 2, 4, 6, 8, 10, 12, 14 (2, 4, 8)	0 θ θ θ 0	$A'E'$
b	0, 2, 16, 18 (2, 16)	θ 0 0 θ 0	$B'C'E'$
c	14, 30 (16)	θ 1 1 1 0	$BCDE'$
d	13, 29 (16)	θ 1 1 0 1	$BCD'E$
e	18, 19 (1)	1 0 0 1 θ	$AB'C'D$
f	12, 13 (1)	0 1 1 0 θ	$A'BCD'$
g	10, 11 (1)	0 1 0 1 θ	$A'BC'D$
h	6, 7 (1)	0 0 1 1 θ	$A'B'CD$

Determination of all Prime Implicants of an Incompletely Specified Function

In case of incompletely specified Boolean function we start with the minterms and don't care terms together for determining all the prime implicants. This is illustrated using the following examples.

Example 1:

$$F(A,B,C,D,E)=\sum(1,2,3,6,8,9,14,17,24,25,26,27,30,31)+\sum_{d.c.}(4,5)$$

The determination of all prime implicants is shown in Table 5.3. The prime implicants are marked from *a* to *l* in Table 5.3 and listed below:

		16 8 4 2 1	
a	26, 27, 30, 31 (1, 4)	1 1 θ 1 θ	ABD
b	24, 25, 26, 27 (1, 2)	1 1 0 θ θ	ABC'
c	8, 9, 24, 25 (1, 16)	θ 1 0 0 θ	$BC'D'$
d	1, 9, 17, 25 (8, 16)	θ θ 0 0 1	$C'D'E$
e	14, 30 (16)	θ 1 1 1 0	$BCDE'$
f	6, 14 (8)	0 θ 1 1 0	$A'CDE'$
g	4, 6 (2)	0 0 1 θ 0	$A'B'CE'$
h	4, 5 (1)	0 0 1 0 θ	$A'B'CD'$
i	2, 6 (4)	0 0 θ 1 0	$A'B'DE'$
j	2, 3 (1)	0 0 0 1 θ	$A'B'C'D$
k	1, 5 (4)	0 0 θ 0 1	$A'B'D'E$
l	1, 3 (2)	0 0 0 θ 1	$A'B'C'E$

Table 5.3 Determination of all prime implicants of the function

$$F(A,B,C,D,E)=\sum(1,2,3,6,8,9,14,17,24,25,26,27,30,31)+\sum_{d.c.}(4,5)$$

Index	Minterms	Stage I	Stage II
1	1 √ 2 √ 4 √ 8 √	1, 3 (2) *l* 1, 5 (4) *k* 1, 9 (8) √ 1, 17 (16) √ 2, 3 (1) *j* 2, 6 (4) *i* 4, 5 (1) *h* 4, 6 (2) *g* 8, 9 (1) √ 8, 24 (16) √	1, 9, 17, 25 (8, 16) *d* 8, 9, 24, 25 (1, 16) *c*
2	3 √ 5 √ 6 √ 9 √ 17 √ 24 √	6, 14 (8) *f* 9, 25 (16) √ 17, 25 (8) √ 24, 25 (1) √ 24, 26 (2) √	24, 25, 26, 27 (1, 2) *b*
3	14 √ 25 √ 26 √	14, 30 (16) *e* 25, 27 (2) √ 26, 27 (1) √ 26, 30, (4) √	26, 27, 30, 31 (1, 4) *a*
4	27 √ 30 √	27, 31 (4) √ 30, 31 (1) √	
5	31 √		

Example 2: $F(A,B,C,D)=\sum(1,3,5,7,9,11,15)+\sum_{d.c.}(4,6,8,14)$

The determination of all prime implicants is shown in Figure 5.4. The prime implicants are marked from *a* to *f* in Table 5.4 and listed below:

		8 4 2 1	
a	6, 7, 14, 15 (1, 8)	θ 1 1 θ	BC
b	3, 7, 11, 15 (4, 8)	θ θ 1 1	CD
c	4, 5, 6, 7 (1, 2)	0 1 θ θ	$A'B$
d	1, 3, 9, 11 (2, 8)	θ 0 θ 1	$B'D$
e	1, 3, 5, 7 (2, 4)	0 θ θ 1	$A'D$
f	8, 9 (1)	1 0 0 θ	$AB'C'$

Table 5.4 Determination of all prime implicants of the function $F(A,B,C,D)=\sum(1,3,5,7,9,11,15)+\sum_{d.c.}(4,6,8,14)$

Index	Minterms	Stage I	Stage II
1	1 √	1, 3 (2) √	1, 3, 5, 7, (2, 4) *e*
	4 √	1, 5 (4) √	1, 3, 9, 11 (2, 8) *d*
	8 √	1, 9 (8) √	4, 5, 6, 7 (1, 2) *c*
-------	----------	4, 5 (1) √	-------------------------
2	3 √	4, 6 (2) √	3; 7, 11, 15 (4, 8) *b*
	5 √	8, 9 (1) *f*	6, 7, 14, 15 (1, 8) *a*
	6 √	------------------	
	9 √	3, 7 (4) √	
-------	----------	3, 11 (8) √	
3	7 √	5, 7 (2) √	
	11 √	6, 7 (1) √	
	14 √	6, 14 (8) √	
-------	----------	9, 11 (2) √	
4	15 √	------------------	
		7, 15 (8) √	
		11, 15 (4) √	
		14, 15 (1) √	

5.5.3 Selection of Prime Implicants

The selection of prime implicants for covering the function consists of the following steps:

1. From the set of all prime implicants, a set of essential prime implicants is determined by making a prime implicant chart.
2. The minterms that are not covered by essential prime implicants are then considered and a minimum cover (secondary-essential prime implicants) is obtained from the remaining prime implicants.

The prime implicant chart is a table. The rows of the table correspond to the prime implicants arranged in order of increasing cost, where *cost* is the number of literals in the prime implicant. The columns correspond to minterms of the function that are to be covered. *If the function has some don't care terms, they are not included in the prime implicant chart*, since they need not to be covered, however, they must be considered for finding out the prime implicants. There is an extra column in the left and an extra row at the bottom. In each row excepting the last row "x" are placed in the columns corresponding to the minterms covered by the prime implicant in that row. The prime implicant chart for the example of Table 5.2 is shown in Table 5.5.

Table 5.5 Prime implicant chart for the example of Table 5.2.

		0	2	4	6	7	8	10	11	12	13	14	16	18	19	29	30
*	*a* 0,2,4,6,8,10,12,14 (2,4,8)	x	x	ⓧ	x		ⓧ	x		x		x					
*	*b* 0,2,16,18 (2,16)	x	x										ⓧ	x			
*	*c* 14,30 (16)											x					ⓧ
*	*d* 13,29 (16)										x					ⓧ	
*	*e* 18,19 (1)													x	ⓧ		
	f 12,13 (1)									x	x						
*	*g* 10,11 (1)							x	ⓧ								
*	*h* 6,7 (1)				x	ⓧ											
		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

After completing the table for all prime implicants of the function, a search is made for the column having a single "x" and they are encircled. In this example we see that there is only one "x" in the columns corresponding to minterms 4, 7, 8, 11, 16, 19, 29, and 30. The prime implicants that cover these minterms are essential prime implicants, since these prime implicants are the only one covering the corresponding minterms. An asterisk is placed in the left most column on the row corresponding to essential prime implicants. We also place checks in the bottom row in columns of the minterms that are covered by the essential prime implicants. The essential prime implicants for this example are *a*, *b*, *c*, *d*, *e*, *g*, and *h*. In this example, we see that all the minterms are covered by the essential prime implicants. The prime implicant *f* is redundant because minterms 12 and 13 are already covered by essential prime implicants. The simplified sum of product expression for the function is, thus, given by

$$F(A,B,C,D,E)=a+b+c+d+e+g+h$$
$$=A'E'+B'C'E'+BCDE'+BCD'E+AB'C'D+A'BC'D+A'B'CD$$

In the above example we see that all the minterms are covered by the essential prime implicants, but it may not be so for all the cases. In general, we are left with some unchecked columns and a set of non-essential prime implicants. The next step is to select the minimum set of secondary-essential prime implicants out of the remaining non-essential prime implicants so as to cover the unchecked minterms. The procedure for this is as follows:

1. Make a reduced prime implicant chart containing non-essential prime implicants and the unchecked minterms.
2. In the reduced prime implicant chart, if two prime implicants cover the same set of minterms, the two corresponding rows are called equivalent rows. We can cancel one prime implicant which is having more cost.
3. We can also cancel the non-dominate row B, where a row A is said to dominate over row B, if row A has "x" in all the columns in which row B has "x" and at the same time A also has "x" in at least one column in which B does not have "x".

4. After canceling the equivalent rows and non-dominate rows, we carry out the same procedure, as we do in selecting essential prime implicants for covering the rest of the minterms.

The above procedure is explained using the prime implicants of Table 5.3. The prime implicant chart is shown in Table 5.6. In this table we make columns for the minterms only.

Table 5.6 Prime implicant chart for the example of Table 5.3.

		1	2	3	6	8	9	14	17	24	25	26	27	30	31
•	*a* 26,27,30,31 (1,4)											x	x	x	ⓧ
	b 24,25,26,27 (1,2)									x	x	x	x		
•	*c* 8,9,24,25 (1,16)					ⓧ	x			x	x				
•	*d* 1,9,17,25 (8,16)	x					x		ⓧ		x				
	e 14,30 (16)							x						x	
	f 6,14 (8)				x			x							
	g 4,6 (2)				x										
	h 4,5 (1)														
	i 2,6 (4)		x		x										
	j 2,3 (1)		x	x											
	k 1,5 (4)	x													
	l 1,3 (2)	x		x											
		✓				✓	✓		✓	✓	✓	✓	✓	✓	✓

It can be seen from the prime implicant chart of Table 5.6 that *a*, *c*, and *d* are essential prime implicants. The reduced prime implicant chart is now made for the remaining minterms and prime implicants in Table 5.7.

Table 5.7 Reduced prime implicant chart for Table 5.6.

		2	3	6	14
	b 24,25,26,27 (1,2)				
	~~*e* 14,30 (16)~~				~~x~~
*	*f* 6,14 (8)			x	ⓧ
	~~*g* 4,6 (2)~~			~~x~~	
	h 4,5 (1)				
	i 2,6 (4)	x		x	
*	*j* 2,3 (1)	x	ⓧ		
	k 1,5 (4)				
	~~*l* 1,3 (2)~~		~~x~~		
		✓	✓	✓	✓

It can be seen from Table 5.7 that prime implicant *e* is dominated by *f*, *g* is dominated by both *f* and *i*, and *l* is dominated by *j*. Therefore, prime implicants *e*, *g*, and *l* are canceled out. Now we look for the columns in which there is a single "x" to find out secondary-essential prime implicants. It can be seen that *f* and *j* are secondary-essential prime implicants and they cover rest of the minterms. The simplified sum of product expression for the function is, thus, given by

$$F(A,B,C,D) = a + c + d + f + j$$
$$= ABD + BC'D' + C'D'E + A'CDE' + A'B'C'D$$

5.5.4 Cyclic Prime Implicant Chart

In some prime implicant charts or sometimes after removing equivalent and non-dominant rows in the reduced prime implicant charts, we may be left with more than one check in each column so that it becomes difficult to select the proper prime implicants. Such prime implicant charts are called *cyclic prime implicant charts* and can be solved by using **Petrick's algebraic procedure**. The procedure is illustrated using the prime implicants for the example of Table 5.4. The prime implicant chart is shown in Table 5.8, which is a cyclic prime implicant chart.

Table 5.8 Cyclic prime implicant chart for example of Table 5.4.

	1	3	5	7	9	11	15
a 6, 7, 14, 15 (1, 8)				x			x
b 3, 7, 11, 15 (4, 8)		x		x		x	x
c 4, 5, 6, 7 (1, 2)			x	x			
d 1, 3, 9, 11 (2, 8)	x	x			x	x	
e 1, 3, 5, 7 (2, 4)	x	x	x	x			
f 8, 9 (1)					x		

From the column for minterm 1, it can be seen that for covering the minterm, either *d* or *e* should be selected. If all the prime implicants are taken as Boolean variables, then we must have $(d+e)=1$. Similarly for covering minterm 3, $(b+d+e)=1$. For covering both the minterms 1 and 3, $(d+e)(b+d+e)=1$. In this way we get the complete expression as

$$(d+e)(b+d+e)(c+e)(a+b+c+e)(d+f)(b+d)(a+b)=1$$

or,

$$(bd+d+de+be+de+e)(ac+bc+c+ce+ae+be+ce+e)$$
$$(bd+d+bf+df)(a+b)=1$$

or, $(d+e)(c+e)(d+bf+df)(a+b)=1$

or, $(cd+de+ce+e)(ad+bd+abf+bf+adf+bdf)=1$

or, $(cd+e)(ad+bd+bf)=1$

or, $acd+bcd+bcdf+ade+bde+bfe=1$

or, $acd+bcd+ade+bde+bfe=1$

Therefore, at least one of the five products in the above equation should be 1. Since all products contain three prime implicants, the final selection will be based on the cost. The costs of prime implicants *a* to *e* is all 2 and the cost of prime implicant *f* is 3. Therefore, the products *acd*, *bcd*, *ade*, and *bde* all cost 6 and the product *bfe* costs 7. As the first four products cost minimum, any of them can be taken for the minimum cover. If we take prime implicants *a*, *c*, and *d* for the minimum cover, the simplified function is

$$F(A,B,C,D) = a + c + d = BC + A'B + B'D$$

Thus, the Petrick's algebraic method gives all possible combinations of prime implicants for covering all the minterms and a combination having minimum number of prime implicants and a minimum cost is selected out of them. For equal cost any combination can be selected.

5.6 Iterated Consensus Method for Simplification of Boolean Function

5.6.1 The Consensus Operation

The consensus for two product terms T_1 and T_2 (denoted $T_1 \Uparrow T_2$) is defined as

(i) If there exists a literal x and product terms T_3 and T_4, which are independent of x, such that $T_1 = x \cdot T_3$ and $T_2 = x' \cdot T_4$, then $T_1 \Uparrow T_2 = T_3T_4$.

(ii) If the above condition is not satisfied, that is no variable appears complemented in one term and uncomplemented in the other, then $T_1 \Uparrow T_2 = 0$.

For example, $x_2'x_4 \Uparrow x_1x_2'x_4' = x_2' \cdot x_1x_2' = x_1x_2'$.

Properties of the Consensus Operation

The consensus operation is commutative but not associative, that is $T_1 \Uparrow T_2 = T_2 \Uparrow T_1$ but $(T_1 \Uparrow T_2) \Uparrow T_3 \neq T_1 \Uparrow (T_2 \Uparrow T_3)$. For example, if $T_1 = x_2'x_4$, $T_2 = x_1x_2'x_4'$, and $T_3 = x_2x_3$ then

$$T_1 \Uparrow T_2 = x_2'x_4 \Uparrow x_1x_2'x_4' = x_2' \cdot x_1x_2' = x_1x_2'$$
$$T_2 \Uparrow T_1 = x_1x_2'x_4' \Uparrow x_2'x_4 = x_1x_2' \cdot x_2' = x_1x_2'$$

but

$$(T_1 \Uparrow T_2) \Uparrow T_3 = (x_2'x_4 \Uparrow x_1x_2'x_4') \Uparrow x_2x_3 = x_1x_2' \Uparrow x_2x_3 = x_1x_3$$
$$T_1 \Uparrow (T_2 \Uparrow T_3) = x_2'x_4 \Uparrow (x_1x_2'x_4' \Uparrow x_2x_3) = x_2'x_4 \Uparrow x_1x_3x_4' = x_1x_2'x_3$$

Another important property of the consensus operation is that $T_1 \Uparrow T_2 \subseteq T_1 + T_2$, which means that the term produced by the consensus of T_1 and T_2 is included in $T_1 + T_2$. For example, if $T_1 = x_2'x_4$ and $T_2 = x_1x_2'x_4'$, then

$$T_1 \Uparrow T_2 = x_1x_2' \subseteq x_2'x_4 + x_1x_2'x_4'$$

5.6.2 Simplification of Boolean Function

The iterated consensus method of simplification of Boolean function depends on the following two steps:

1. Determine all prime implicants of the function by the repeated application of the consensus operation.
2. Determine a minimum cover of the function using the methods of sections 5.5.3 and 5.5.4.

Determination of All Prime Implicants

Unlike Quine-McCluskey method, the iterated consensus method does not require that the function be represented as sum of minterms. The simplification can be done using any sum of products form. The prime implicants of the function is determined using the following steps:

Step 1: Make a list of product terms included in the function.

Step 2: For each pair of terms, T_i and T_j (including the new terms added to the list) find out $T_i \Uparrow T_j$ and add it to the list if $T_i \Uparrow T_j \neq 0$.

Each time a term is added to the list, check out the terms which are equal to or included in the new term. Do not use any checked term for finding new consensus product. Care must be taken to insure that each unchecked term has been compared with all other unchecked terms in the list. For this purpose, starting from the second term, each unchecked term in the list is compared with all the unchecked terms above it. The process ends when all possible consensus operations have been performed.

Step 3: The unchecked terms in the list are the prime implicants of the function.

The procedure is illustrated for the function $f(x_1,x_2,x_3,x_4) = x_2'x_4 + x_1x_2'x_4' + x_1'x_2x_4 + x_1'x_2'x_3'$.

Step 1: The product terms of the function are listed above the horizontal line in Table 5.9.

Step 2: The outcome of step 2 is shown below the horizontal line in Table 5.9. Consensus operations of possible pairs are shown and necessary explanations are given in the right of each operation.

Table 5.9 Determination of prime implicants of the function $f(x_1,x_2,x_3,x_4)=x_2'x_4+x_1x_2'x_4'+x_1'x_2x_4+x_1'x_2'x_3'$ using iterated consensus method.

Step 1: $T_1 : x_2'x_4$ √

$T_2 : x_1x_2'x_4'$ √

$T_3 : x_1'x_2x_4$ √

$T_4 : x_1'x_2'x_3'$ √

Step 2: $T_5 : x_1x_2'$ $T_2 \Uparrow T_1 = x_1x_2' = T_5 \supseteq T_2$ (T_5 is added to the list and T_2 is checked out)

$T_6 : x_1'x_4$ $T_3 \Uparrow T_1 = x_1'x_4 = T_6 \supseteq T_3$ (T_6 is added to the list and T_3 is checked out)

$T_4 \Uparrow T_1 = 0$

$T_5 \Uparrow T_1 = 0$

$T_7 : x_2'x_3'$ $T_5 \Uparrow T_4 = x_2'x_3' = T_7 \supseteq T_4$ (T_7 is added to the list and T_4 is checked out)

$T_6 \Uparrow T_1 = 0$

$T_8 : x_2'x_4$ $T_6 \Uparrow T_5 = x_2'x_4 = T_8 = T_1$ (T_8 is added to the list and T_1 is checked out)

$T_7 \Uparrow T_5 = 0$

$T_7 \Uparrow T_6 = 0$

$T_8 \Uparrow T_5 = 0$

$T_8 \Uparrow T_6 = 0$

$T_8 \Uparrow T_7 = 0$

Step 3: The prime implicants of the function are x_1x_2', $x_1'x_4$, $x_2'x_3'$, and $x_2'x_4$.

Selection of prime Implicants

The selection of prime implicants is done by the methods of sections 5.5.3 and 5.5.4. However, for constructing the prime implicant chart, if the function is not already in the sum of minterms form, it is expanded into sum of minterms form. This can easily be done using K-map if the function is of 4 or less variables. If the function is of more than 4 variables, algebraic method can be used. The expansion of the function $f(x_1,x_2,x_3,x_4)=x_2'x_4+x_1x_2'x_4'+x_1'x_2x_4+x_1'x_2'x_3'$ using K-map is shown in Figure 5.21, which gives

$$f(x_1,x_2,x_3,x_4)=x_2'x_4+x_1x_2'x_4'+x_1'x_2x_4+x_1'x_2'x_3'$$

$x_1x_2 \backslash x_3x_4$	00	01	11	10
00	1	1	1	
01		1	1	
11				
10	1	1	1	1

$$f(x_1,x_2,x_3,x_4)=x_2'x_4+x_1x_2'x_4'+x_1'x_2x_4+x_1'x_2'x_3'$$
$$=\sum(0,1,3,5,7,8,9,10,11)$$

Figure 5.21 Expansion of $f(x_1,x_2,x_3,x_4)=x_2'x_4+x_1x_2'x_4'+x_1'x_2x_4+x_1'x_2'x_3'$ into sum of minterms form.

In the prime implicant chart, the prime implicants and the minterms are represented using their binary notations, for example, the minterm 0 is represented as 0000 and the prime implicant x_1x_2' is represented as 10—. A prime implicant covers a minterm if the non-missing literals of the prime implicant are exactly the same as those in the minterm. For example, the prime implicant 10— covers the minterms 1000, 1001, 1010, and 1011. The prime implicant chart for the example of Table 5.9 is shown in Table 5.10. In Table 5.10, the essential prime implicants are x_1x_2', $x_1'x_4$, and $x_2'x_3'$, which covers all the minterms. Therefore, the simplified function is

$$f(x_1,x_2,x_3,x_4)=x_1x_2'+x_1'x_4+x_2'x_3'$$

Table 5.10 Prime implicant chart for the example of Table 5.9

		0 0000	1 0001	3 0011	5 0101	7 0111	8 1000	9 1001	10 1010	11 1011
*	x_1x_2' (10−−)						x	x	ⓧ	x
*	$x_1'x_4$ (0−−1)		x	x	ⓧ	ⓧ				
*	$x_2'x_3'$ (−00−)	ⓧ	x				x	x		
	$x_2'x_4$ (−0−1)		x	x				x		x
		√	√	√	√	√	√	√	√	√

The advantage of the iterated consensus method over Quine-McCuskey method is that usually fewer comparisons are needed to be made. Moreover, it is not necessary to expand the function to canonical sum of products form and can be started with any partially simplified function.

Exercises

5.1 Simplify algebraically the following functions.

(a) $F = (A' + B)(A + B + D)D'$

(b) $F = A'BC + AB'C + ABC' + ABC$

(c) $F = A'BCD + AB'C'D' + AB'C'D + AB'CD' + AB'CD +$
$ABC'D' + ABC'D + ABCD' + ABCD$

(d) $F = ABC + A'C$

(e) $F = (A + B)(A' + B')$

(f) $F = (ABC)'(A + B + C)'$

(g) $F = (B + C')(B' + C) + (A' + B + C')'$

(h) $F = (C + D)' + A'CD' + AB'C' + A'B'CD + ACD'$

5.2 (a) Show that a group of 2^n 1s on an n-variable K-map produces a constant 1.

(b) Show that a group of 2^n 0s on an n-variable K-map produces a constant 0.

5.3 Simplify the following sum of products functions using K-map method.

(a) $F(A, B, C, D) = \sum(0,1,2,3,7,8,9,10,11,12,13)$

(b) $F(A, B, C, D) = \sum(0,2,4,6,8) + \sum_{d.c.}(10,11,12,13,14,15)$

(c) $F(A, B, C, D) = \sum(0,2,4,9,12,15) + \sum_{d.c.}(1,5,7,10)$

(d) $F(A, B, C, D) = \sum(0,3,4,5,6,7,8,13,14)$

(e) $F(A, B, C, D) = \sum(4,6,7,9,10,11,12,14,15)$

(f) $F(A, B, C, D) = \sum(0,1,2,3,4,6,7,8,9,11,15)$

(g) $F(A, B, C, D) = \sum(1,3,4,5,7,8,9,11,14,15)$

(h) $F(A, B, C, D) = \sum(1,2,3,4,5,6,13,14,15)$

(i) $F(A, B, C, D) = ABC'D + A'BCD + A'B'C' + A'B'D' +$
$AC' + AB'C + B'$

(j) $F(A, B, C, D) = \sum(1,3,7,11,15) + \sum_{d.c.}(0,2,5)$

(k) $f(x, y, z) = x'yz + x'yz' + xy'z' + xy'z$

(l) $F(A, B, C) = A'C + A'B + AB'C + BC$

(m) $F(A, B, C, D) = A'B'C' + B'CD' + A'BCD' + AB'C'$

(n) $F(A, B, C, D) = \sum(3,4,5,7,9,13,14,15)$

(o) $F(A, B, C, D) = \sum(0,1,2,3,5,7,8,9,11,14)$

5.4 Simplify the following product of sums functions using K-map method.

(a) $F(A, B, C, D) = \prod(1,4,5,6,11,12,13,14,15)$

(b) $F(A, B, C, D) = \prod(4,5,6,7,8,12) \cdot \prod_{d.c.}(1,2,3,9,11,14)$

(c) $F(A, B, C, D) = (A + B + C' + D')(A' + C + D')(A' + B + C' + D')$
$(B' + C)(B' + C')(A + B')(B' + D')$

5.5 Simplify the following functions using Quine-McCluskey method.

(a) $F(A, B, C, D, E) = \sum(1,3,6,10,11,12,14,15,17,19,20,22,24,29,30)$

(b) $F(A, B, C, D, E) = \sum(1,3,6,7,8,12,14,16,18,22,23,24,28,30,31)$

(c) $F(A, B, C, D, E) = \sum(0,1,4,5,6,7,9,11,15) + \sum_{d.c.}(10,14)$

(d) $F(A, B, C, D, E) = \sum(4,8,10,16,21,27,28) + \sum_{d.c.}(1,5,23,25,30,31)$

(e) $F(A, B, C, D, E) = \sum(1,5,6,12,13,14) + \sum_{d.c.}(2,4)$

(f) $F(A, B, C, D) = \sum(1,2,3,5,13) + \sum_{d.c.}(6,7,8,9,11,15)$

(g) $F(A, B, C, D, E) = \sum(0,1,2,3,4,6,9,10,15,16,17,18,20,23,25,26,31)$

(h) $F(A, B, C, D, E) = \sum(1,5,6,7,9,13,14,15,17,18,19,21,22,23,25,29,30)$

(i) $F(A, B, C, D, E) = \sum(8,12,13,18,19,21,22,24,25,28,31) +$
$\sum_{d.c.}(1,2,3,6,7,11)$

5.6 Simplify the following functions using iterated consensus method.

(a) $f(x_1, x_2, x_3, x_4) = x_1'x_2'x_3' + x_1'x_3x_4 + x_1'x_2x_4 + x_1x_2'$

(b) $f(x_1, x_2, x_3, x_4) = x_1'x_2x_3' + x_1x_2x_4 + x_2x_3x_4 + x_1x_2'x_3 + x_1'x_2'x_3x_4'$

(c) $F(A, B, C, D) = A'B'C' + BC'D' + A'CD + ACD + ACD'$

(d) $F(A, B, C, D) = A'C'D' + BC'D + B'C'D + ABC' + A'BD' + BCD'$

(e) $F(A, B, C, D, E) = \sum(0,1,4,5,6,7,9,11,15) + \sum_{d.c.}(10,14)$

(f) $F(A, B, C, D) = \sum(0,3,4,5,6,7,8,13,14)$

Further Reading

1. E. J. McCluskey, *Logic Design Pronciples*, Prentice-Hall, Inc., 1986.
2. Zvi Kohavi, *Switching and Finite Automata Theory*, 2nd ed., McGraw-Hill Higher Education, 1990.
3. Richard S. Sandige, *Modern Digital Design*, McGraw-Hill Publishing Company, 1990.
4. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
5. D. C. Green, *Digital Electronics*, 5th ed., Pearson Education, 2001.
6. V. K. Jain, *Switching Theory and Digital Electronics*, Khanna Publishers, India, 2002.

CHAPTER 6
COMBINATIONAL LOGIC DESIGN

6.1 Introduction

Digital logic circuits may be combinational or sequential. A *combinational* circuit consists of logic gates whose outputs at any time are determined directly from the present combination of inputs without regard to previous inputs. A combinational circuit performs a specific information processing operations fully specified logically by a set of Boolean functions. *Sequential* circuits employ memory element in addition to logic gates. Their outputs are a function of the inputs and the state of the memory elements. The state of memory elements, in turn, is a function of previous inputs. As a consequence, the outputs of a sequential circuit depend not only on present inputs, but also on past inputs, and the circuit behavior must be specified by a time sequence of inputs and internal states. Sequential circuits are discussed in Chapter 10.

The purpose of this chapter is to use the knowledge acquired in previous chapters and formulate systematic design and analysis procedures of combinational circuits.

A combinational circuit consists of input variables, logic gates, and output variables. The logic gates accept signals from the inputs and generate signals to the outputs. This process transforms binary information from the given input data to the required output data. Obviously, both input and output are represented by binary signals, i.e., they exist in two possible values, one representing 1 and the other 0. A block diagram of a combinational circuit is shown in Figure 6.1. The n input binary variables come from an external source; the m output variables go to an external destination. In many applications, the source and/or destination are storage registers (discussed in Chapter 11) located either in the vicinity of the combinational circuit or in a remote external device.

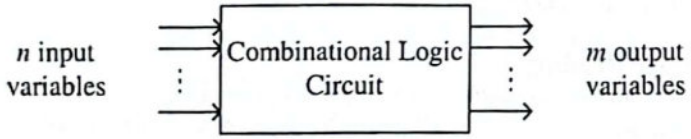


Figure 6.1 Block diagram of a combinational circuit.

For n input variables, there are 2^n possible combinations of binary input values. For each possible input combination, there is one and only one possible output combination. A combinational circuit can be described by m Boolean functions, one for each output variable. Each output function is expressed in terms of the n input variables.

6.2 Design of Combinational Circuits

The design of combinational circuits involves the following steps:

1. The problem is stated.

2. The number of available input variables and required output variables is determined. The input and output variables are assigned letter symbols and logic values.
3. The truth table that defines the required relationships between inputs and outputs is derived.
4. The simplified Boolean function for each output is obtained.
5. The logic diagram is obtained.

A truth table for a combinational circuit consists of input columns and output columns. The binary values for the outputs are determined from examination of the stated problem. The output functions specified in the truth table give the exact definition of the combinational circuit. It is important that the verbal specifications be interpreted correctly into a truth table. Sometimes the designer must use his intuition and experience to arrive at the correct interpretation. Word specifications are very seldom complete and exact. Any wrong interpretation that results in an incorrect truth table produces a combinational circuit that will not fulfil the stated requirements.

The output Boolean functions from the truth table are simplified by any available method, such as algebraic simplification, the K-map method, the Quine-McCluskey method, or the iterated consensus method. Usually there will be a variety of simplified expressions from which to choose. However, in any particular application, certain restrictions, limitations, and criteria will serve as a guide in the process of choosing a particular algebraic expression. A practical design method would have to consider such conditions as

(1) minimum number of gates,
(2) minimum number of inputs to a gate,
(3) minimum propagation time of the signal through the circuit,
(4) minimum number of interconnections, and
(5) limitations of the driving capabilities of each gate.

Since all these criteria can not be satisfied simultaneously, and since the importance of each constraint is dictated by the particular application, it is difficult to make a general statement as to what constitutes an acceptable simplification. In most cases the simplification begins by satisfying an elementary objectives, such as producing a simplified Boolean function in a standard form, and from that proceeds to meet any other performance criteria.

In practice, designers tend to go from the Boolean functions to a wring list that shows the interconnections among various standard logic gates. In that cases the design need not go any further than the required simplified output Boolean functions. However, a logic diagram is helpful for visualizing the gate implementation of the expressions.

The design procedure is demonstrated using the following example.

1. **The problem statement:** The door of an explosive-warehouse is operated on by four persons, namely, A, B, C, and D. Person A along with at least any one from B, C, and D can open the door. If A is absent, B, C, and D can open the

door altogether. A combinational logic circuit is to be designed for controlling the door.

2. Four input lines are required for four keys operated by four persons. The four input variables are assigned letter symbols *A*, *B*, *C*, and *D* after the names of the four persons. The input variables are assigned the following logic values:

    The key is operated for opening the door: 1
    The key is operated for closing the door: 0

    One output line is required for controlling the lock of the door. The output variable is assigned the letter symbol *X* and the following logic values:

    The lock is opened: 1
    The lock is closed: 0

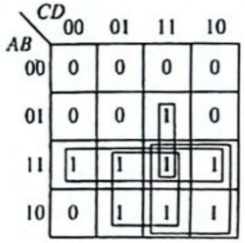
3. The truth table defining the relationship between inputs and output is derived and shown in Table 6.1.

Table 6.1 Truth table for explosive-warehouse door control example.

A B C D	*X*
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	0
0 1 0 0	0
0 1 0 1	0
0 1 1 0	0
0 1 1 1	1
1 0 0 0	0
1 0 0 1	1
1 0 1 0	1
1 0 1 1	1
1 1 0 0	1
1 1 0 1	1
1 1 1 0	1
1 1 1 1	1

4. The simplified Boolean function for the truth table of Table 6.1 is obtained by K-map method in Figure 6.2, which gives

$$X = AB + AC + AD + BCD$$



AB \ CD	00	01	11	10
00	0	0	0	0
01	0	0	1	0
11	1	1	1	1
10	0	1	1	1

Figure 6.2 Simplification of the function of Table 6.1 by K-map.

5. The logic diagram of the door control circuit is shown in Figure 6.3.

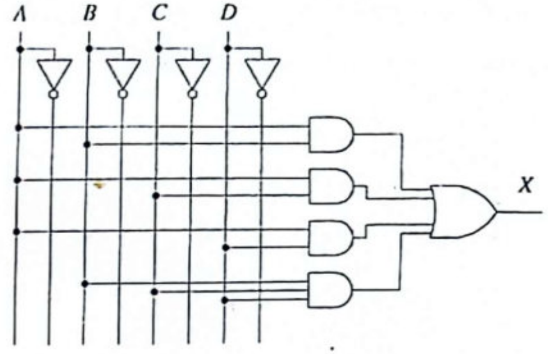


Figure 6.3 Logic diagram for the explosive-warehouse door control example.

The design of some typical elementary functions important for the understanding of digital computers and systems will be discussed in the following chapters.

6.3 Analysis of Combinational Circuits

The design of a combinational circuit starts from the verbal specifications of a required function and culminates with a logic diagram. The analysis of a combinational circuit is somewhat the reverse process. It starts with a given logic diagram and culminates with a set of output Boolean functions or a truth table or a verbal explanation of the circuit operation. If the logic diagram to be analyzed is accompanied by a function name or an explanation of what it is assumed to accomplish, then the analysis problem reduces to a verification of the stated function.

The first step in the analysis is to make sure that the given circuit is combinational and not sequential. The diagram of a combinational circuit has logic gates with no feedback paths or memory elements. A feedback path is a connection from the output of one gate to the input of a second gate that forms part of the input to the first gate. Feedback paths or memory elements in a digital circuit define a sequential circuit and must be analyzed according to procedures outlined in Chapter 10.

Derivation of Boolean Expressions

Once the logic diagram is verified as a combinational circuit, the Boolean expressions can be derived using the following steps:

1. Label with arbitrary symbols all gate outputs that are a function of the external input variables only. Obtain the Boolean function for each gate.
2. Label with other arbitrary symbols those gates that are a function of input variables and/or previously labeled gates. Find the Boolean functions for these gates. Repeat the above process until the outputs of the circuit are obtained.

3. By repeated substitution of previously defined functions, obtain the output Boolean functions in terms of external input variables only.

The procedure is demonstrated below for the example combinational circuit of Figure 6.4.

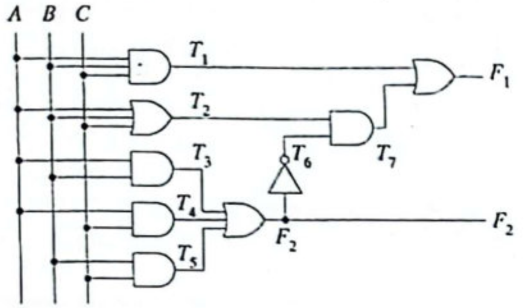


Figure 6.4 Combinational logic circuit for analysis example.

1. The outputs of gates that are a function of external input variables only are T_1, T_2, T_3, T_4, and T_5. The Boolean functions for these five outputs are

$$T_1 = ABC$$
$$T_2 = A+B+C$$
$$T_3 = AB$$
$$T_4 = AC$$
$$T_5 = BC$$

2. Now we consider outputs of gates that are a function of already defined symbols:

$$F_2 = T_3 + T_4 + T_5$$
$$T_6 = F_2'$$
$$T_7 = T_2T_6$$
$$F_1 = T_1 + T_7$$

3. The output Boolean function F_2 in terms of external input variables only is obtained as follows:

$$F_2 = T_3 + T_4 + T_5 = AB + AC + BC$$

The output Boolean function F_1 in terms of external input variables only is obtained by repeated substitution of previously defined functions as follows:

$$\begin{aligned} F_1 &= T_1 + T_7 \\ &= ABC + T_2T_6 \\ &= ABC + (A+B+C)F_2' \\ &= ABC + (A+B+C)(AB+AC+BC)' \\ &= ABC + (A+B+C)(A'+B')(A'+C')(B'+C') \\ &= ABC + (AB'+A'B+A'C+B'C)(A'B'+A'C'+B'C'+C') \\ &= ABC + AB'C' + A'BC' + A'B'C \end{aligned}$$

Derivation of Truth Table

Construction of truth table is straightforward from the derived Boolean functions. However, the truth table can be obtained directly from the logic diagram without going through the derivations of the Boolean functions by using the following steps:

1. Determine the number of input variables to the circuit. For n inputs, form the 2^n possible input combinations of 1s and 0s.
2. Label the outputs of gates with arbitrary symbols.
3. Obtain the truth table for the outputs of those gates that are a function of external input variables only.
4. Proceed to obtain the truth table for the outputs of those gates that are a function of previously defined values until the columns for all outputs are determined.

This process is demonstrated using the circuit of Figure 6.4 as follows:

1. The input variables of the circuit are A, B, and C. The $2^3 = 8$ input combinations of 1s and 0's for three variables are shown in the left column of Table 6.2.
2. The gates are labeled as in Figure 6.4.
3. Outputs T_1, T_2, T_3, T_4, and T_5 are functions of external input variables only and they are

$$T_1 = ABC$$
$$T_2 = A+B+C$$
$$T_3 = AB$$
$$T_4 = AC$$
$$T_5 = BC$$

Truth table for T_1, T_2, T_3, T_4, and T_5 are derived from input combinations and are shown in the Table 6.2.

4. Outputs F_2, T_6, T_7, and F_1 are function of previously defined values and they are

$$F_2 = T_3 + T_4 + T_5$$
$$T_6 = F_2'$$
$$T_7 = T_2 T_6$$
$$F_1 = T_1 + T_7$$

Truth table for F_2 is derived from the truth table of T_3, T_4, and T_5. Truth table for T_6 is derived from the truth table of F_2. Truth table for T_7 is derived from the truth tables of T_2 and T_6. Finally, truth table for F_1 is derived from the truth tables of T_1 and T_7. These truth tables are shown in Table 6.2.

Table 6.2 Truth table for the operation of the logic diagram of Figure 6.4.

ABC	T_1	T_2	T_3	T_4	T_5	F_2	T_6	T_7	F_1
000	0	0	0	0	0	0	1	0	0
001	0	1	0	0	0	0	1	1	1
010	0	1	0	0	0	0	1	1	1
011	0	1	0	0	1	1	0	0	0
100	0	1	0	0	0	0	1	1	1
101	0	1	0	1	0	1	0	0	0
110	0	1	1	0	0	1	0	0	0
111	1	1	1	1	1	1	0	0	1

6.4 Two-Level NAND Circuits

Digital circuits are more frequently constructed with NAND and NOR gates than with AND and OR gates. NAND and NOR gates are easier to fabricate with electronic components and are the basic gates in all IC digital logic families. Because of the prominence of NAND and NOR gates in the design of digital circuits, rules and procedures have been developed for the conversion from Boolean functions given in terms of AND, OR, and NOT into equivalent NAND or NOR logic diagrams.

The implementation of a Boolean function with NAND gates requires that the function be simplified in the sum of products form. To see the relationship between a sum of products expression and its equivalent NAND implementation, consider the logic diagram drawn in Figure 6.5. All three diagrams are equivalent and implement the function

$$F = A + BC + CD$$

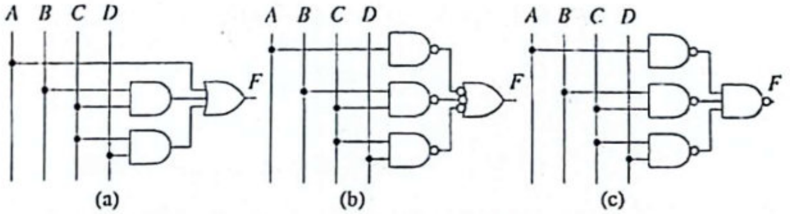


Figure 6.5 Three ways to implement $F = A + BC + CD$.

The function is implemented in Figure 6.5(a) in sum of products form with AND and OR gates. In Figure 6.5(b), the AND gates are replaced by NAND gates and the OR gate is replaced by a NAND gate with an invert-OR symbol (see Figure 4.8). The single variable A is complemented using one-input NAND gate (see Figure 4.8) and applied to the second-level invert-OR gate. Remember that a small circle denotes complementation. Therefore, two circles on the same line represents double complementation and both can be removed. The complement of A goes through a small circle which complements the variable again to produce the uncomplemented value of A. Removing the small circles in the gates of Figure 6.5(b) produces the circuit in Figure 6.5(a). Therefore, the two diagrams implement the same function and are equivalent. In Figure 6.5(c), the output NAND gate is redrawn with the conventional symbol. The diagram in Figure 6.5(c) is equivalent to the one in Figure 6.5(b), which in turn is equivalent to the diagram in Figure 6.5(a). Note the similarity between the diagrams in Figure 6.5(a) and 6.5(c). The AND and OR gates have been changed to NAND gates and an additional one-input NAND gate has been included with the single variable A.

The NAND implementation in Figure 6.5(c) can be verified algebraically using DeMorgan's theorem as follows:

$$F = (A'(BC)'(CD)')' = A + BC + CD$$

From the transformation shown in Figure 6.5, we conclude that a Boolean function can be implemented with two levels of NAND gates. The rule for obtaining the NAND logic diagram from a Boolean function is as follows:

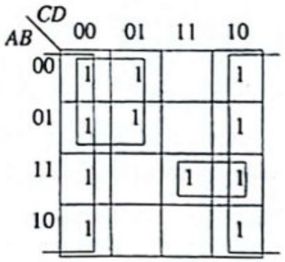
1. Simplify the function and express it in sum of products form.
2. Draw a NAND gate for each product term of the function that has at least two literals. The inputs to each NAND gate are the literals of the product term. This constitutes a group of first-level gates. A term with a single literal requires an inverter in the first level. Therefore, complement a single literal and apply as an input to the second level.
3. Draw a single NAND gate in the second level, with inputs coming from the first level.

The procedure is demonstrated for the following function

$$F(A,B,C,D) = \sum(0,1,2,4,5,6,8,10,12,14,15)$$

Step 1. The simplification of the function as sum of products using K-map is shown in Figure 6.6, which gives

$$F(A,B,C,D) = D' + A'C' + ABC$$



AB \ CD	00	01	11	10
00	1	1		1
01	1	1		1
11	1		1	1
10	1			1

Figure 6.6 Simplification of $F(A,B,C,D) = \sum(0,1,2,4,5,6,8,10,12,14,15)$ in sum of products form using K-map.

Steps 2 and 3. The two-level NAND implementation of the simplified function using steps 2 and 3 is shown in Figure 6.7.

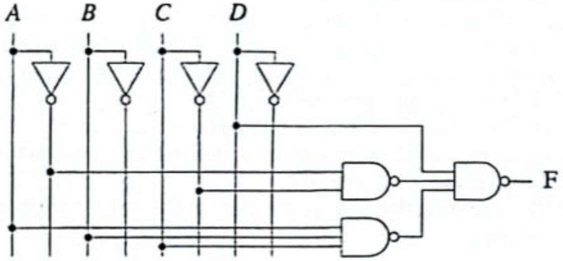


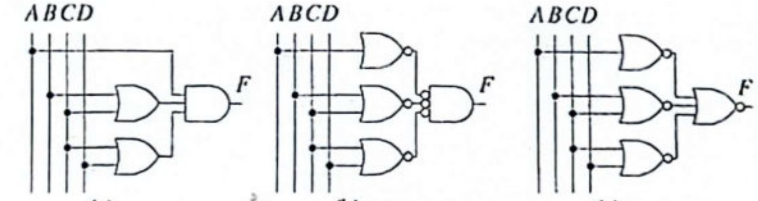
Figure 6.7 Two-level NAND implementation of $F(A,B,C,D) = D' + A'C' + ABC$.

6.5 Two-Level NOR Circuits

The NOR function is the dual of the NAND function. For this reason, all procedures and rules for NOR logic are the dual of the corresponding procedures and rules developed for NAND logic.

The implementation of a Boolean function with NOR gates requires that the function be simplified in product of sums form. A product of sums expression specifies a group of OR gates for the sum terms, followed by an AND gate to produce the product. The transformation from the OR-AND to the NOR-NOR diagram is depicted in Figure 6.8 for the product of sums expression

$$F = A(B + C)(C + D)$$



(a) (b) (c)

Figure 6.8 Three way to implement $F = A(B + C)(C + D)$.

The rule for obtaining the NOR logic diagram from a Boolean function can be derived from this transformation and is as follows:

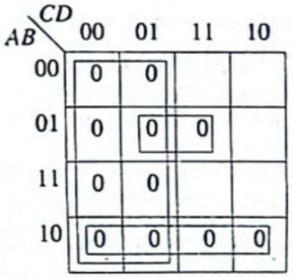
1. Simplify the function and express it in product of sums form.
2. Draw a NOR gate for each sum term of the function that has at least two literals. The inputs to each NOR gate are the literals of the sum term. This constitutes a group of first-level gates. A term with a single literal requires an inverter in the first level. Therefore, complement a single literal and apply as an input to the second level.
3. Draw a single NOR gate in the second level, with inputs coming from the first level.

The procedure is demonstrated for the function

$$F(A,B,C,D) = \prod(0,1,4,5,7,8,9,10,11,12,13)$$

Step 1. The simplification of the function as product of sums form using K-map is shown in Figure 6.9, which gives

$$F(A,B,C,D) = C(A' + B)(A + B' + D')$$



AB \ CD	00	01	11	10
00	0	0		
01	0	0	0	
11	0	0		
10	0	0	0	0

Figure 6.9 Simplification of $F(A,B,C,D) = \prod(0,1,4,5,7,8,9,10,11,12,13)$ in product of sums form using K-map.

Steps 2 and 3. The two-level NOR implementation of the simplified function is shown in Figure 6.10.

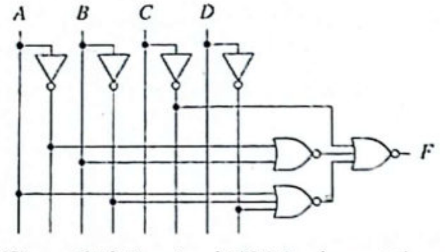


Figure 6.10 Two-level NOR implementation of $F(A,B,C,D) = C(A'+B)(A+B'+D')$.

6.6 Multi-Level NAND Circuits

6.6.1 Boolean Function Implementation with Multi-Level NAND Circuit

The implementation of Boolean functions with multi-level NAND circuits may be obtained by means of a simple block diagram manipulation technique. This method requires that two other logic diagrams be drawn prior to obtaining the NAND logic diagram. Nevertheless, the procedure is very simple and straightforward:

1. From the given algebraic expression, draw the logic diagram with AND, OR, and NOT gates. Assume that both the uncomplemented and complemented inputs are available.
2. Draw a second logic diagram with the equivalent NAND logic, as given in Figure 4.6, substituted for each AND, OR, and NOT gate.
3. Remove any two cascaded inverters from the diagram, since double inversion does not perform a logic function. Remove inverters connected to single external inputs and complement the corresponding input variable. The new logic diagram obtained is the required NAND gate implementation.

This procedure is demonstrated using the following examples.

Example 1: The function is $F = B(C + AD) + CD'$.

1. The AND-OR implementation of this function is shown in the logic diagram of Figure 6.11(a).
2. For each AND gate, we substitute a NAND gate followed by an inverter; for each OR gate, we substitute input inverters followed by a NAND gate. This substitution follows directly from the logic equivalence of Figure 4.6 and is shown in the diagram of Figure 6.11(b). This diagram has seven inverters (excluding the inverters used for indicating the complemented input) and five two-input NAND gates listed with numbers inside the gate symbol.
3. Pairs of inverters connected in cascade (from each AND box to each OR box) are removed since they form double inversion. The inverter connected to input C is removed and the input is connected to C' input. The result is the NAND logic diagram shown in Figure 6.11(c), with the number inside each symbol identifying the gate from Figure 6.11(b).

This example demonstrated that the number of NAND gates required to implement the Boolean function is equal to the number of AND-OR gates, provided both the uncomplemented and the complemented inputs are available.

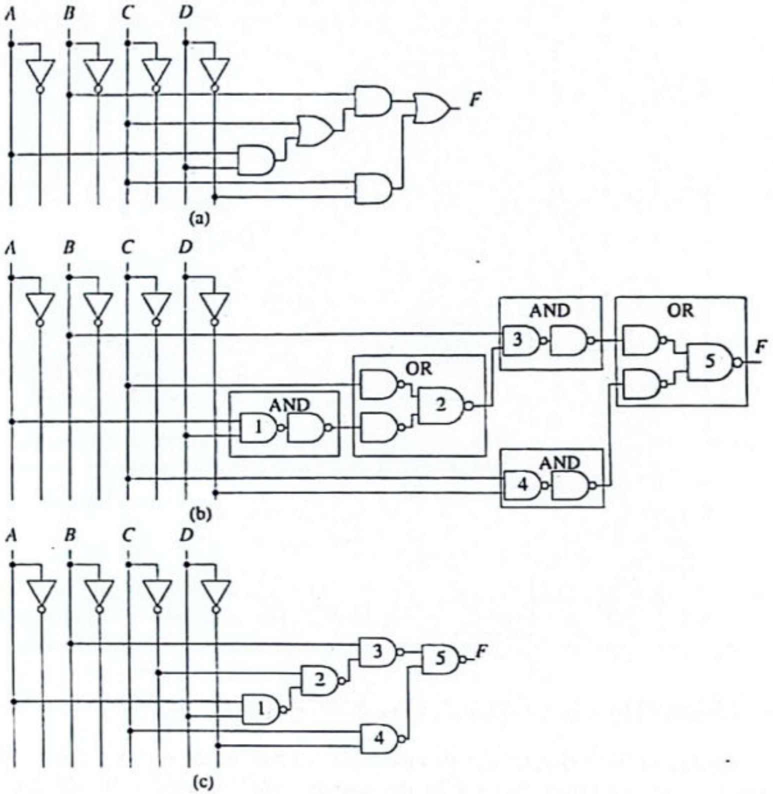


Figure 6.11 Implementation of $F = B(C + AD) + CD'$ with NAND gates.

Example 2: The function is $F = (A' + C)(BC + D)$.

1. The AND-OR implementation is shown in Figure 6.12(a).
2. The NAND logic substitution is shown in Figure 6.12(b).
3. One pair of cascaded inverters from gate 1 to gate 3 is removed. The three external inputs A', C, and D, which directly go to inverters, are complemented and the corresponding inverters are removed. The final NAND gate implementation is shown in Figure 6.12(c).

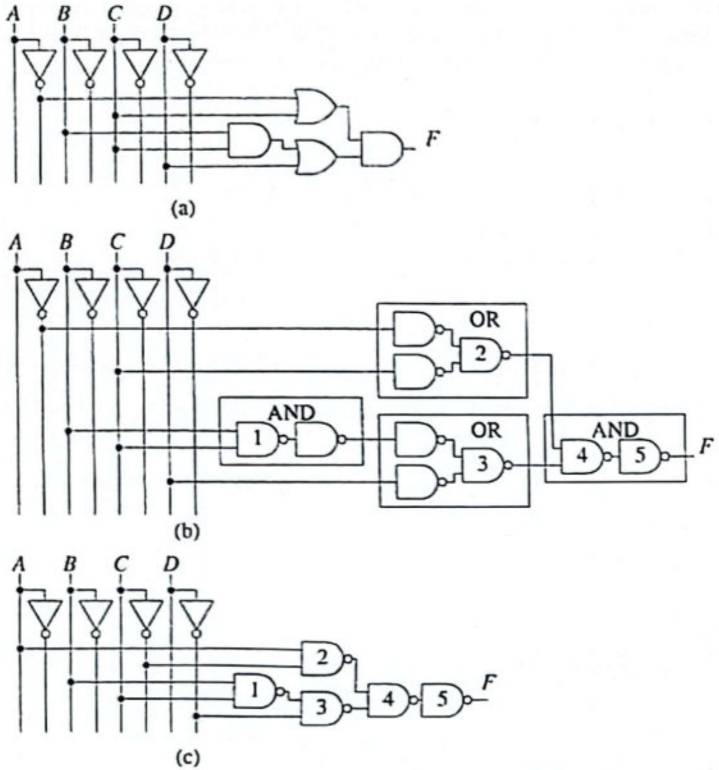


Figure 6.12 Implementation of $F = (A' + C)(BC + D)$ with NAND gates.

The number of NAND gates for this example is equal to the number of AND-OR gates plus an additional inverter in the output (NAND gate 5). In general, the number of NAND gates required to implement a function equals the number of AND-OR gates, except for an occasional inverter. This is true provided both the uncomplemented and the complemented inputs are available.

The block diagram method is somewhat tiresome to use because it requires the drawing of two logic diagrams to obtain the answer in a third.

6.6.2 Analysis of Multi-Level NAND Circuits

The foregoing procedure considered the problem of deriving a NAND logic diagram from a given Boolean function. The reverse process is the analysis problem, which starts with a given NAND logic diagram and culminates with a Boolean expression or a truth table. The analysis of NAND logic diagrams follows the same procedure presented in Section 6.3. The only difference is that NAND logic requires a repeated application of DeMorgan's theorem.

Derivation of the Boolean Expression by Algebraic Manipulation

The procedure for deriving the Boolean expression from a logic diagram is outlined in Section 6.3. This procedure is demonstrated for the NAND logic diagram of Figure 6.11(c), which is redrawn in Figure 6.13.

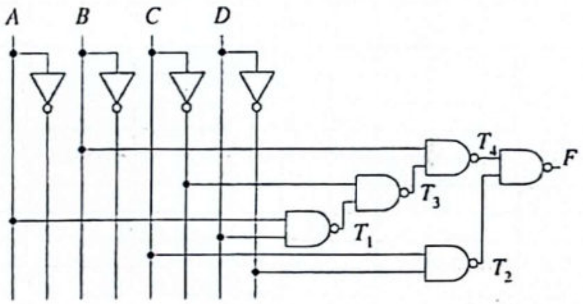


Figure 6.13 Analysis example of NAND circuit.

First, all gate outputs are labeled with arbitrary symbols. Second, the Boolean functions for the outputs of gates that receive only external inputs are derived:

$$T_1 = (AD)' = A' + D'$$
$$T_2 = (CD')' = C' + D$$

The second form follows directly from DeMorgan's theorem and may, at times, be more convenient to use. Third, Boolean functions of gates which have inputs from previously derived functions are determined in consecutive order until the output is expressed in terms of external input variables:

$$T_3 = (C'T_1)' = (A'C' + C'D')' = (A + C)(C + D) = AC + AD + C + CD$$
$$= C + AD$$
$$T_4 = (BT_3)' = (B(C + AD))'$$
$$F = (T_4T_2)' = ((B(C + AD))'(CD')')' = B(C + AD) + CD'$$

Derivation of the Truth Table

The procedure for obtaining the truth table directly from a logic diagram is also outlined in Section 6.3. This procedure is demonstrated for the NAND logic diagram of Figure 6.13. First, the four input variables, together with their 16 combinations of 1s and 0s, are listed in Table 6.3. Second, the outputs of all gates are labeled with arbitrary symbols as in Figure 6.13. Third, we obtain the truth table for the outputs of those gates that are a function of the external input variables only. These are T_1 and T_2. $T_1 = (AD)'$, so we mark 0s in those rows where both A and D are equal to 1 and fill the rest of the rows of T_1 with 1s. Also, $T2 = (CD')'$, so we mark 0s in those rows where $C = 1$ and $D = 0$, and fill the rest of the rows of T_2 with 1s.

We then proceed to obtain the truth table for the outputs of those gates that are a function of previously defined outputs until the column for the output F is determined.

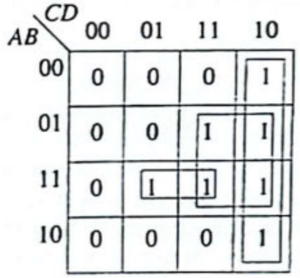
Table 6.3 Truth table for the circuit of Figure 6.13.

$ABCD$	T_1	T_2	T_3	T_4	F
0 0 0 0	1	1	0	1	0
0 0 0 1	1	1	0	1	0
0 0 1 0	1	0	1	1	1
0 0 1 1	1	1	1	1	0
0 1 0 0	1	1	0	1	0
0 1 0 1	1	1	0	1	0
0 1 1 0	1	0	1	0	1
0 1 1 1	1	1	1	0	1
1 0 0 0	1	1	0	1	0
1 0 0 1	0	1	1	1	0
1 0 1 0	1	0	1	1	1
1 0 1 1	0	1	1	1	0
1 1 0 0	1	1	0	1	0
1 1 0 1	0	1	1	0	1
1 1 1 0	1	0	1	0	1
1 1 1 1	0	1	1	0	1

It is now possible to obtain an algebraic expression for the output from the derived truth table. The K-map shown in Figure 6.14 is obtained directly from Table 6.3. The simplified expression obtained from the K-map is

$$F = BC + ABD + CD' = B(C + AD) + CD'$$

This is the same as the expression of Figure 6.11, thus verifying the correct answer.



AB \ CD	00	01	11	10
00	0	0	0	1
01	0	0	1	1
11	0	1	1	1
10	0	0	0	1

Figure 6.14 Derivation of F from Table 6.3 using K-map.

Conversion of NAND Logic Diagram to AND-OR Logic Diagram

It is sometimes convenient to convert a NAND logic diagram to its equivalent AND-OR logic diagram to facilitate the analysis procedure. By doing so, the Boolean function can be derived more easily without employing DeMorgan's theorem. The conversion of logic diagram is accomplished through a process reverse from that used for implementation. In Figure 4 8, we showed two alternate graphic symbols for the NAND gate. By judicious use of both symbols, it is possible to convert a NAND diagram to an equivalent AND-OR form.

The conversion of a NAND logic diagram to an AND-OR diagram is achieved through a change in symbols from AND-invert to invert-OR in alternate levels of gates. The first level to be changed to an invert-OR symbol should be the last level. These changes produce pairs of circles along the same line, and these can be removed since they represent double complementation. Moreover, a one-input AND or OR gate can be removed since it does not perform a logical function. A one-input AND or OR with a circle in the input or output is changed to an inverter.

This procedure is demonstrated in Figure 6.15. The NAND logic diagram of Figure 6.15(a) is that of Figure 6.11(c) and is to be converted to an AND-OR logic diagram. The symbol of the gate in the last level is changed to an invert-OR. Looking for alternate levels, we find one gate requiring a change of symbol as shown in Figure 6.15(b). Any two circles along the same line are removed. Circles that go to external inputs are also removed and the corresponding input variable is complemented. The required AND-OR logic diagram is drawn in Figure 6.15(c).

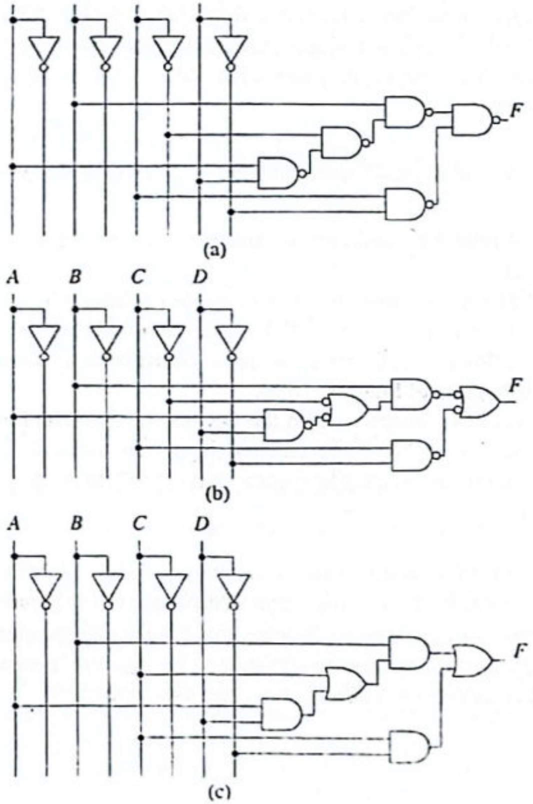


Figure 6.15 Conversion of NAND logic diagram to AND-OR logic diagram

6.7 Multi-Level NOR Circuits

The NOR function is the dual of the NAND function. For this reason, procedures and rules for multi-level NOR logic form a dual of the corresponding procedures and rules developed for multi-level NAND logic.

6.7.1 Boolean Function Implementation with Multi-Level NOR Circuit

The block diagram procedure for implementing Boolean function with multi-level NOR circuits is similar to the procedure outlined in the previous section for NAND gates.

1. From the given algebraic expression, draw the logic diagram with AND, OR, and NOT gates. Assume that both the uncomplemented and the complemented inputs are available.
2. Draw a second logic diagram with the equivalent NOR logic, as given in Figure 4.7, substituted for each AND, OR, and NOT gate.
3. Remove any two cascaded inverters from the diagram. Remove inverters connected to single external inputs and complement the corresponding input variable. The new logic diagram obtained is the required NOR gate implementation.

The procedure is illustrated for the function $F = B(C + AD) + CD'$.

1. The AND-OR implementation of this function is shown in the logic diagram of Figure 6.16(a).
2. For each AND gate, we substitute input inverters followed by a NOR gate; for each OR gate, we substitute a NOR gate followed by an inverter. This substitution follows directly from the logic equivalence of Figure 4.7 and is shown in the diagram of Figure 6.16(b).
3. The pair of cascaded inverters from the OR box to the AND box is removed. The five inverters connected to external inputs are removed and the input variables are complemented. The results is the NOR logic diagram shown in Figure 6.16(c).

The number of NOR gates in this example equals the number of AND-OR gates plus an additional inverter in the output (NOR gate 6). In general, the number of NOR gates required to implement a Boolean function equals the number of AND-OR gates, except for an occasional inverter. This is true provided both the uncomplemented and the complemented inputs are available.

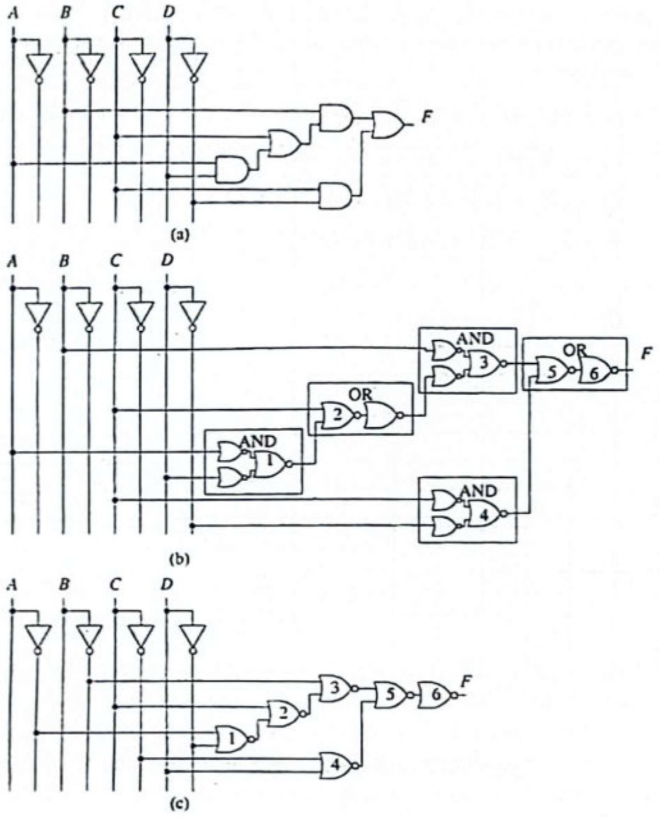


Figure 6.16 Implementation of $F = B(C + AD) + CD'$ with NOR gates.

6.7.2 Analysis of Multi-Level NOR Circuits

The analysis of NOR logic diagram follows the same procedure presented in Section 6.3.

Derivation of the Boolean Expression by Algebraic Manipulation

The procedure for deriving the Boolean function from a NOR logic diagram is demonstrated for the logic diagram of Figure 6.16(c), which is redrawn in Figure 6.17. First, all gate outputs are labeled with arbitrary symbols. Second, the Boolean functions for the outputs of gates that receive only external inputs are derived:

$$T_1 = (A' + D')' = AD$$
$$T_2 = (C' + D)' = CD'$$

Third. Boolean functions of gates which have inputs from previously derived functions are determined in consecutive order until the output is expressed in terms of input variables:

$$T_3 = (C + T_1)' = (C + AD)'$$
$$T_4 = (B' + T_3)' = (B' + (C + AD)')' = B(C + AD)$$
$$T_5 = (T_4 + T_2)' = (B(C + AD) + CD')'$$
$$F = T_5' = B(C + AD) + CD'$$

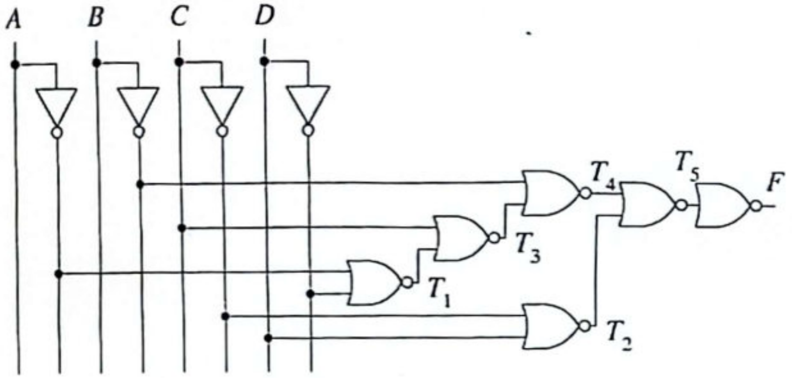


Figure 6.17 Analysis example of NOR circuit.

Derivation of the Truth Table

To obtain the truth table directly from the logic diagram of Figure 6.17, first the four input variables, together with their 16 combinations of 1s and 0s, are listed in Table 6.4. Second, the outputs of all gates are labeled with arbitrary symbols as in Figure 6.17. Third, we obtain the truth table for the outputs of those gates that are a function of the external input variables only. These are T_1 and T_2. $T_1 = (A' + D')'$, so we mark 0s in those rows where $A = 0$ or $D = 0$ and fill the rest of the rows of T_1 with 1s. Also, $T_2 = (C' + D)'$, so we mark 0s in those rows where $C = 0$ or $D = 1$, and fill the rest of the rows of T_2 with 1s. We then proceed to obtain the truth table for the outputs of those gates that are a function of previously defined outputs until the column for the output F is determined.

Table 6.4 Truth table for the circuit of Figure 6.17.

$ABCD$	T_1	T_2	T_3	T_4	T_5	F
0 0 0 0	0	0	1	0	1	0
0 0 0 1	0	0	1	0	1	0
0 0 1 0	0	1	0	0	0	1
0 0 1 1	0	0	0	0	1	0
0 1 0 0	0	0	1	0	1	0
0 1 0 1	0	0	1	0	1	0
0 1 1 0	0	1	0	1	0	1
0 1 1 1	0	0	0	1	0	1
1 0 0 0	0	0	1	0	1	0
1 0 0 1	1	0	0	0	1	0
1 0 1 0	0	1	0	0	0	1
1 0 1 1	1	0	0	0	1	0
1 1 0 0	0	0	1	0	1	0
1 1 0 1	1	0	0	1	0	1
1 1 1 0	0	1	0	1	0	1
1 1 1 1	1	0	0	1	0	1

The truth table for F obtained in Table 6.4 is the same as the truth table of F in Table 6.3. The algebraic expression can be obtained as in Figure 6.14.

Conversion of NOR Logic Diagram to AND-OR Logic Diagram

To convert a NOR logic diagram to its equivalent AND-OR logic diagram, we use the two symbols for NOR gate shown in Figure 4.8. The OR-invert is the normal symbol for a NOR gate and the invert-AND is a convenient alternative that utilizes DeMorgan's theorem and the convention that small circles at the inputs denote complementation.

The conversion of a NOR logic diagram to an AND-OR diagram is achieved through a change in symbols from OR-invert to invert-AND starting from the last level and in alternate levels. Pairs of small circles along the same line are removed. A one-input AND or OR gate is removed, but if it has a small circle at the input or output, it is converted to an inverter.

This process is demonstrated in Figure 6.18, where the NOR logic diagram in Figure 6.18(a) is converted to an AND-OR diagram. The symbol of the gate in the last level (level 5) is changed to an invert-AND. Looking for alternate levels, we find two gates in level 3 and one in level 1. These three gates undergo a symbol change as shown in Figure 6.18(b). Any two circles along the same line are removed. Circles that go to external inputs are also removed and the corresponding input variable is complemented. The gate in level 5 becomes a one-input AND gate and is removed. The required AND-OR logic diagram is drawn in Figure 6.18(c).

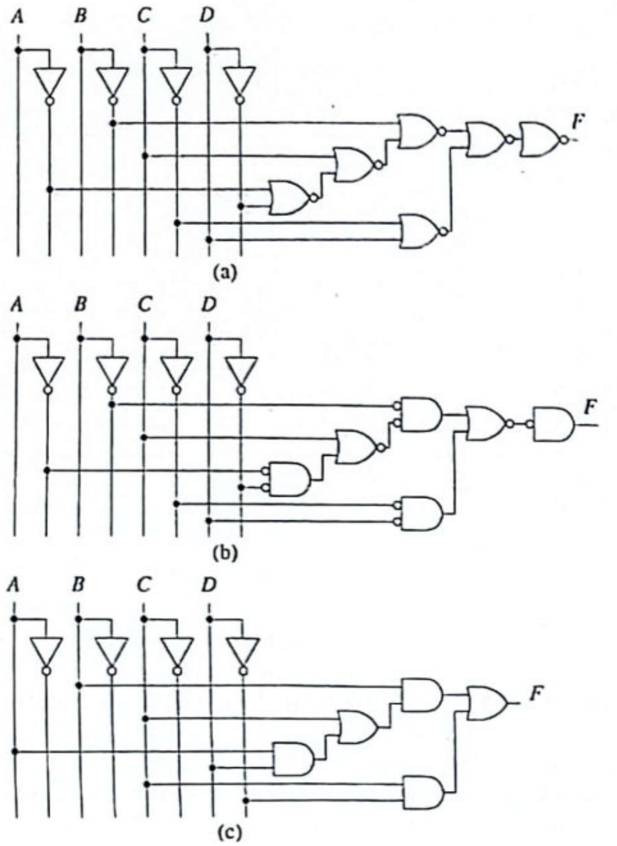


Figure 6.18 Conversion of NOR logic diagram to AND-OR logic diagram.

6.8 Two-Level AND-EXOR Logic Circuit

In conventional logic design a Boolean function is simplified in a sum of products form and the simplified function is implemented using two-level AND-OR circuit. It has been observed that logic implementation with EXOR gates exhibits many fold advantages over conventional OR-based logic implementation. One of such advantages to be noted is that, in many cases, the number of AND gates and the total number of input connections required for EXOR-based logic implementation is less than or at most equal to the number of AND gates and the total number of input connections required for OR-based logic implementation.

The OR and EXOR functions are introduced in Chapter 4. The truth tables for OR and EXOR functions are given in Figure 4.1 and are repeated in Table 6.5 for convenience.

Table 6.5 Truth table for OR and EXOR functions

$x\,y$	$F = x + y$	$F = x \oplus y$
0 0	0	0
0 1	1	1
1 0	1	1
1 1	1	0

From Table 6.5, we observe that OR and EXOR functions are the same for the second and the third rows of the truth table, that is, OR and EXOR functions are the same for exclusive inputs. Therefore, if A and B are two exclusive inputs or two exclusive product terms, we can write $A + B = A \oplus B$. Similarly, if A, B, and C are mutually exclusive product terms, we can write $A + B + C = A \oplus B \oplus C$.

A canonical sum of products (CSOP) function is a sum of minterms. From Table 3.6 it can be seen that all the minterms are mutually exclusive, that is, at a time only one of the minterms can be present in the function. Therefore, the OR symbols of a canonical sum of products function can be replaced by EXOR symbols. This type of function is called canonical exclusive-OR sum of products (CESOP) function. For example, the function $F(A, B, C, D) = \sum(0,1,3,6,7,11)$ can be expressed as

$$\begin{aligned} F(A,B,C,D) &= A'B'C'D' + A'B'C'D + A'B'CD + A'BCD' + A'BCD + AB'CD \\ &= A'B'C'D' \oplus A'B'C'D \oplus A'B'CD \oplus A'BCD' \oplus A'BCD \oplus AB'CD \end{aligned}$$

As another example, consider the function

$$F(A, B, C, D) = A'CD' + A'C'D + AB'C'$$

The product terms $A'CD'$, $A'C'D$, and $AB'C'$ are mutually exclusive, that is, only one product term can be present at a time in the function. Therefore, the function can be written as

$$F(A, B, C, D) = A'CD' + A'C'D + AB'C' = A'CD' \oplus A'C'D \oplus AB'C'$$

A set of product terms are *mutually exclusive* if all possible pairs of product terms of the set are pair wise exclusive. Two product terms are *pair wise exclusive* if their product results in a 0 value. For example, $A'CD' \cdot A'C'D = 0$. Therefore, $A'CD'$ and $A'C'D$ are pair wise exclusive. On a K-map, the groups of 1s representing a set of mutually exclusive product terms are non-overlapping.

Simplification of Exclusive-OR Sum of Products Function

A canonical exclusive-OR sum of products function is an exclusive-OR sum of minterms. These minterms can be grouped on a K-map for simplification of exclusive-OR sum of products (ESOP) function. From the properties of OR function, we know that $A + A + \cdots + A = A$. For this property, we can include a

1 in many overlapping groups of 1s during simplification of sum of products function using K-map. But the simplification of exclusive-OR sum of products function using K-map is different. From the properties of EXOR function, we know that $A \oplus A = 0$ but $A \oplus A \oplus A = A$. This property of EXOR function restricts us to include a 1 in only odd number of overlapping groups of 1s. This property also allows us to include a 0 within a group of 1s if that 0 is included even number of times. An even number of inclusion of a 0 will have no effect on the value of the function.

A two-level AND-EXOR circuit can be designed using the following procedure.

1. Represent the function on a K-map. Identify largest possible groups of cells such that a 1 is included odd number of times and a 0 is included even number of times.
2. Read the product terms for each group of cells. Combine the product terms by EXOR operators to form the simplified exclusive-OR sum of products function.
3. Draw the logic diagram for the minimized function.

The procedure is demonstrated using the following examples.

Example 1. The function is $F(A,B,C,D) = \sum(0,1,3,6,7,11)$.

1. The function is represented on the K-map of Figure 6.19(a). The groups of cells are identified as groups 1, 2, and 3. The cell 0010 containing 0 is included in groups 1 and 2, that is, two times. The cell 0011 containing 1 is included in all three groups, that is, three times.
2. The product terms for group 1 is $A'B'$; for group 2 is $A'C$; and for group 3 is $B'CD$. The simplified exclusive-OR sum of products function is

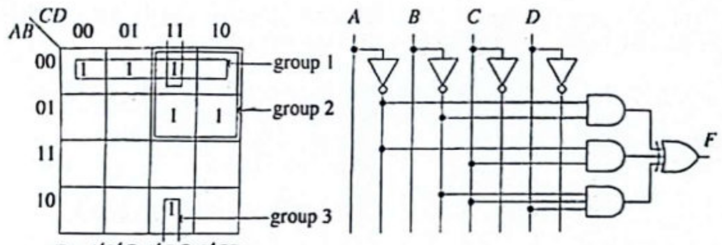
$$F(A,B,C,D) = A'B' \oplus A'C \oplus B'CD$$

3. The AND-EXOR logic diagram is drawn in Figure 6.19(b).

The corresponding sum of products simplification of the function is shown in Figure 6.19(c), which gives

$$F(A,B,C,D) = A'B'C' + B'CD + A'BC$$

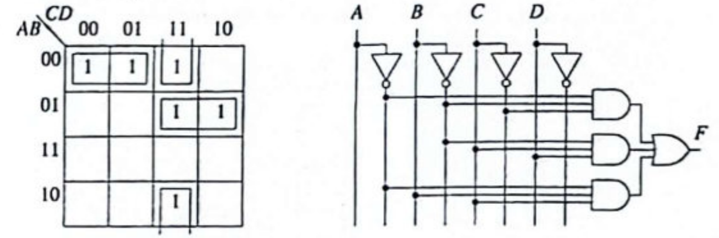
The AND-OR logic diagram is drawn in Figure 6.19(d). Comparing logic diagrams of Figures 6.19(b) and (d), we see that both implementations require three AND gates, but the AND-EXOR implementation requires a total of 7 input connections, whereas the AND-OR implementation requires a total of 9 input connections.



$F = A'B' \oplus A'C \oplus B'CD$

(a) ESOP simplification

(b) AND-EXOR implementation



$F = A'B'C' + B'CD + A'BC$

(c) SOP simplification

(d) AND-OR implementation

Figure 6.19 AND-EXOR and AND-OR implementations of the function $F(A,B,C,D) = \sum(0,1,3,6,7,11)$.

Example 2. The function is $F(A,B,C,D) = \sum(0,1,3,6,7,8,10,13)$.

1. The function is represented on the K-map of Figure 6.20(a). The groups of cells are identified as groups 1, 2, 3, and 4. The cell 0010 containing 0 is included in groups 1 and 2; the cell 0100 containing 0 is included in groups 1 and 3; the cell 0101 containing 0 is included in groups 1 and 4; and the cell 0000 containing 1 is included in groups 1, 2, and 3.
2. The product term for group1 is A'; for group 2 is $B'D'$; for group 3 is $A'C'D'$; and for group 4 is $BC'D$. The simplified exclusive-OR sum of products function is

$$F(A,B,C,D) = A' \oplus B'D' \oplus A'C'D' \oplus BC'D$$

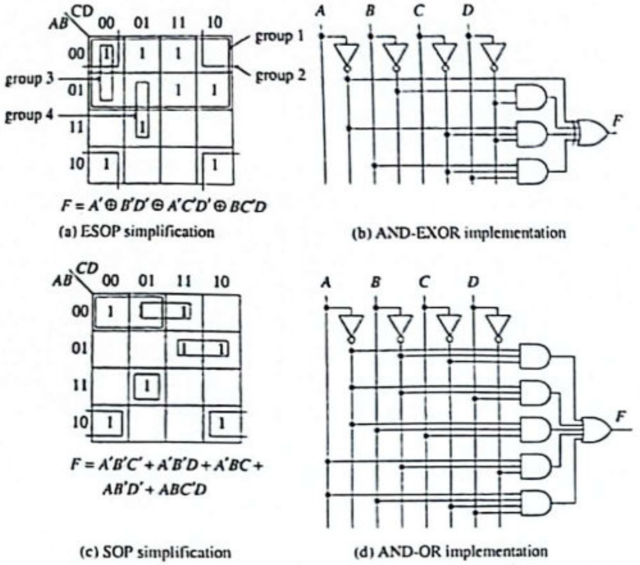
3. The AND-EXOR logic diagram is drawn in Figure 6.20(b).

The corresponding sum of products simplification of the function is shown in Figure 6.20(c), which gives

$$F(A,B,C,D) = A'B'C' + A'B'D + A'BC + AB'D' + ABC'D$$

The AND-OR logic diagram is drawn in Figure 6.20(d). Comparing logic diagrams of Figures 6.20(b) and (d), we see that AND-EXOR implementation requires three

AND gates and a total of 9 input connections, whereas the AND-OR implementation requires five AND gates and a total of 16 input connections.



$F = A' \oplus B'D' \oplus A'C'D' \oplus BC'D$

(a) ESOP simplification

(b) AND-EXOR implementation

$F = A'B'C' + A'B'D + A'BC + AB'D' + ABC'D$

(c) SOP simplification

(d) AND-OR implementation

Figure 6.20 AND-EXOR and AND-OR implementations of the function $F(A,B,C,D) = \sum(0,1,3,6,7,8,10,13)$.

6.9 Three-Level AND-OR-EXOR Logic Circuits

In many cases, AND-EXOR circuits require fewer AND gates and fewer input connections than AND-OR circuits. However, from hardware point of view, multi-input EXOR gate is more expensive than multi-input OR gate. Therefore, it is desirable to limit the input of EXOR gates to as minimum as possible. This is achieved by introducing OR gates between AND gates and EXOR gate. The following procedure converts an AND-EXOR circuit into a three-level AND-OR-EXOR circuit without increasing the number of AND gates.

1. Partition the product terms of the exclusive-OR sum of products function into groups of mutually exclusive product terms. Replace the EXOR operators of each mutually exclusive group by OR operators.
2. If possible, simplify the newly formed sum of products expression.
3. Draw logic diagram using AND, OR, and EXOR gates.

The procedure is demonstrated using the following examples.

Example 1. The exclusive-OR sum of products function is $F(A,B,C,D) = A'C \oplus AD \oplus AB' \oplus BC'D \oplus B'CD'$.

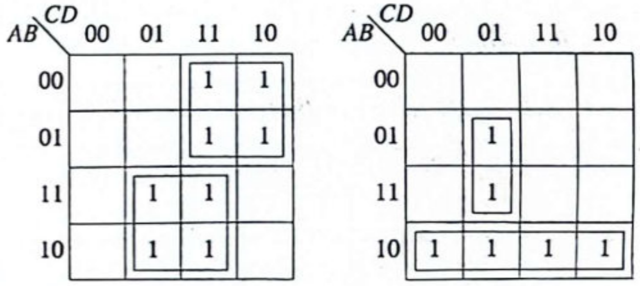
1. The product terms of the exclusive-OR sum of products function are partitioned into groups of mutually exclusive product terms as follows

$$F(A,B,C,D) = (A'C \oplus AD) \oplus (AB' \oplus BC'D) \oplus B'CD'$$

After replacing the EXOR operators of the mutually exclusive groups of product terms, the function becomes

$$F(A,B,C,D) = (A'C + AD) \oplus (AB' + BC'D) \oplus B'CD'$$

2. The newly formed sum of products expression can not be simplified further as shown in the K-maps of Figure 6.21.



(a) $A'C + AD$ (B) $AB' + BC'D$

Figure 6.21 Simplification of (a) $A'C + AD$ and (b) $AB' + BC'D$.

3. The AND-OR-EXOR logic diagram is drawn in Figure 6.22.

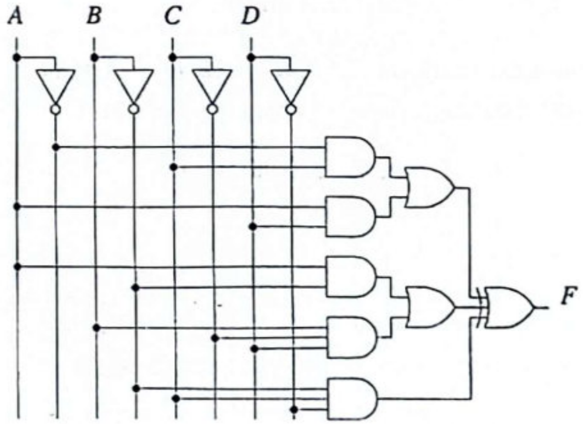


Figure 6.22 AND-OR-EXOR logic diagram for the function $F(A,B,C,D) = A'C \oplus AD \oplus AB' \oplus BC'D \oplus B'CD'$ $= (A'C + AD) \oplus (AB' + BC'D) \oplus B'CD'$

Example 2. The exclusive-OR sum of products function is $F(A,B,C,D) = A'B' \oplus AB'D' \oplus ABCD' \oplus B'C$.

1. The product terms of the exclusive-OR sum of products function are partitioned into groups of mutually exclusive product terms as follows

$$F(A,B,C,D) = (A'B' \oplus AB'D' \oplus ABCD') \oplus B'C$$

After replacing the EXOR operators of the mutually exclusive group of product terms, the function becomes

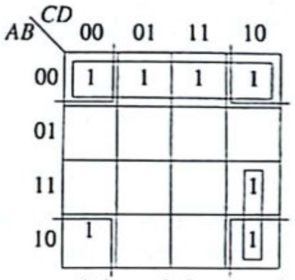
$$F(A,B,C,D) = (A'B' + AB'D' + ABCD') \oplus B'C$$

2. The newly formed sum of products expression $A'B' + AB'D' + ABCD'$ can be simplified further as shown in the K-map of Figure 6.23, which gives

$$A'B' + AB'D' + ABCD' = A'B' + B'D' + ACD'$$

Therefore, the function becomes

$$F(A,B,C,D) = (A'B' + B'D' + ACD') \oplus B'C$$



$A'B' + AB'D' + ABCD'$

$= A'B' + B'D' + ACD'$

Figure 6.23 Simplification of $A'B' + AB'D' + ABCD'$.

3. The AND-OR-EXOR logic diagram is drawn in Figure 6.24.

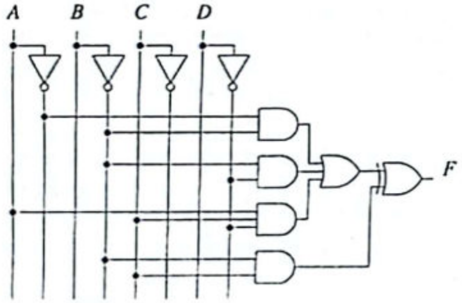


Figure 6.24 AND-OR-EXOR logic diagram for the function $F(A,B,C,D) = A'B' \oplus AB'D' \oplus ABCD' \oplus B'C = (A'B' + B'D' + ACD') \oplus B'C$

Exercises

6.1 Design a combinational logic circuit that has three inputs A, B, and C, and whose output will be high only when a majority of the inputs is high.

6.2 In a combinational logic circuit, four input lines A, B, C, and D are being used to represent a 4-bit binary number with A as the MSB and D as the LSB. The circuit produces a high output only when the binary number is greater than 0110_2. Design this circuit.

6.3 Design a combinational logic circuit with inputs A, B, and C so that output F is high whenever $A = 0$ or whenever $B = C = 1$.

6.4 A combinational circuit has four inputs and one output. The output is equal to 1 when (1) all the inputs are equal to 1 or (2) none of the inputs are equal to 1 or (3) an odd number of inputs are equal to 1.

6.5 Design a combinational circuit that accepts a three-bit number and generates an output binary number equal to the square of the input number.

6.6 Design a combinational circuit with four input lines that represent a decimal digit in BCD and four output lines that generates the 9's complement of the input digit.

6.7 Design a combinational circuit whose input is a four-bit number and whose output is the 2's complement of the input number.

6.8 Design a combinational circuit that detects an error in the representation of a decimal digit in BCD. In other words, obtain a logic diagram whose output is 1 when the inputs contain an unused combination in the code.

6.9 Analyze the two-output combinational circuit shown in Figure 6.25.

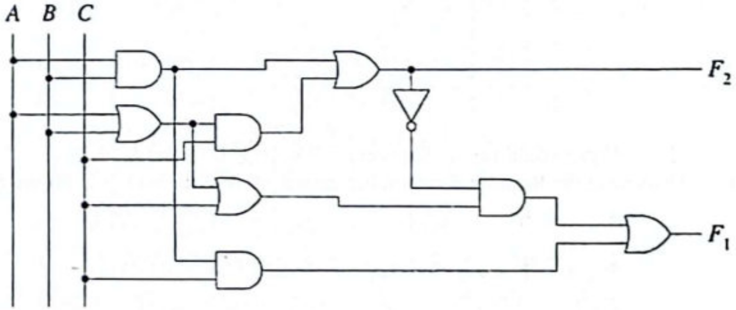


Figure 6.25 Logic diagram for analysis problem 6.9.

6.10 Simplify each of the following functions and implement them with NAND gates.

(a) $F(A,B,C,D,E) = AC' + ACE + ACE' + A'CD' + A'D'E'$

(b) $F(A,B,C,D) = (B' + D')(A' + C' + D)(A + B' + C' + D)(A' + B + C' + D')$

6.11 Repeat problem 6.10 for NOR implementation.

6.12 Implement the following functions with NAND gates. Assume that both the uncomplemented and the complemented inputs are available.

(a) $F = BD + BCD + AB'C'D' + A'B'CD'$

(b) $F = (AB + A'B')(CD' + C'D)$

6.13 Implement the following function with NOR gates. Assume that both the uncomplemented and the complemented inputs are available.

(a) $F = AB' + C'D' + A'CD' + C'D(AB + A'B') + BD(AC' + A'C)$

(b) $F = AB'CD' + A'BCD' + AB'C'D + A'BC'D$

6.14 Implement the following function with both NAND and NOR gates. Only the uncomplemented inputs are available.

$f = w'xz + w'yz + x'yz' + wxy'z$

$d = wyz$

Here, *d* represents the don't care terms.

6.15 Using the block diagram method, convert the logic diagram of Figure 6.26 to a NAND implementation.

6.16 Using the block diagram method, convert the logic diagram of Figure 6.26 to a NOR implementation.

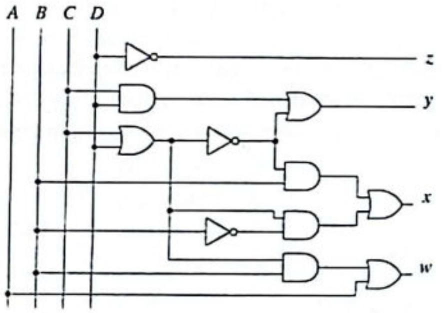


Figure 6.26 Logic diagram for Problems 6.15 and 6.16.

6.17 Determine the Boolean function for the output *F* of the circuit in Figure 6.27.

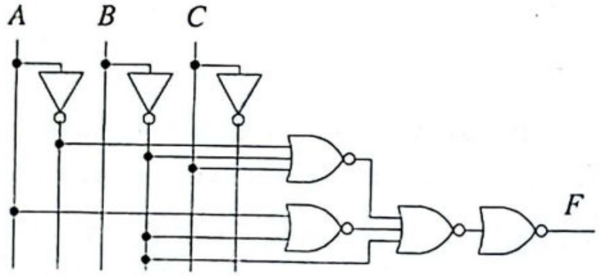


Figure 6.27 Logic diagram for Problem 6.17.

6.18 Determine the output Boolean functions of the circuits in Figure 6.28.

6.19 Obtain the truth tables for the circuits in Figure 6.28.

6.20 Obtain the equivalent AND-OR logic diagrams of Figure 6.28.

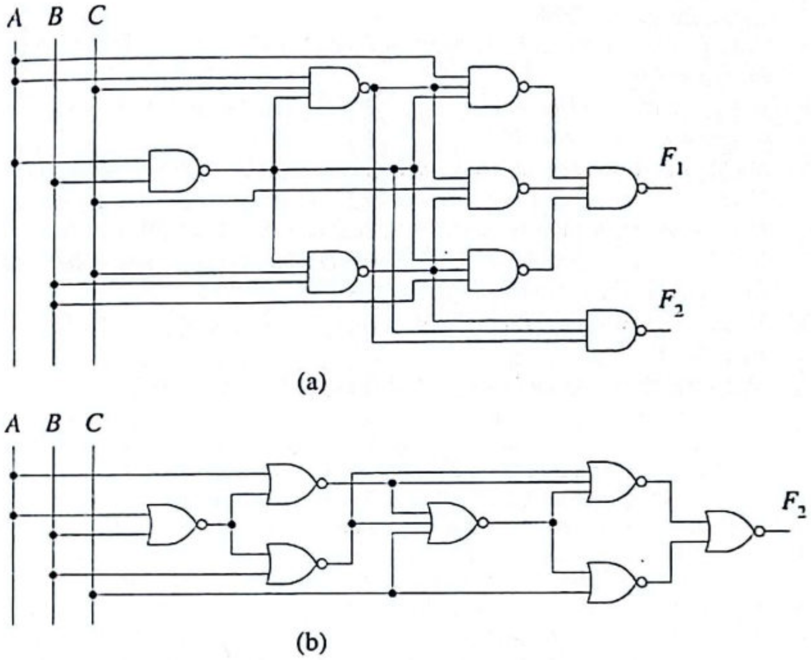


Figure 6.28 Logic diagrams for Problems 6.18, 6.19, and 6.20.

6.21 Design AND-EXOR circuits for the following functions

(a) $F(A,B,C,D) = \sum(0,1,8,11)$

(b) $F(A,B,C,D) = \sum(1,2,4,7,9,10,13,14)$

(c) $F(A,B,C,D) = \sum(0,1,3,6,7,8,10,13)$

(d) $F(A,B,C,D) = \sum(0,1,2,3,8,11)$

(e) $F(A,B,C,D) = \sum(0,1,3,6,7,8,11)$

6.22 Design AND-OR-EXOR circuits for the AND-EXOR circuits obtained in Problem 6.21.

Further Reading

1. E. J. McCluskey, *Logic Design Pronciples*, Prentice-Hall, Inc., 1986.
2. Zvi Kohavi, *Switching and Finite Automata Theory*, 2nd ed., McGraw-Hill Higher Education, 1990.
3. Tsutomu Sasao (editor), *Logic Synthesis and Optimization*, Kluwer Academic Publishers, 1993.
4. R. P. Jain, *Modern Digital Electronics*, 2nd ed., Tata McGraw-Hill Publishing Company Limited, India, 1998.
5. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
6. D. C. Green, *Digital Electronics*, 5th ed., Pearson Education, 2001.
7. Ronald J. Tocci and Neal S. Widmer, *Digital Systems: Principles and Applications*, 8th ed., Prentice-Hall, Inc., 2001.
8. V. K. Jain, *Switching Theory and Digital Electronics*, Khanna Publishers, India, 2002.
9. M. Morris Mano, *Digital Design*, 3rd ed., Pearson Education, 2002.

CHAPTER 7
ARITHMETIC AND COMPARATOR CIRCUITS

7.1 Introduction

Digital components perform a variety of information-processing tasks. Among the basic functions encountered are the various arithmetic operations and comparison. Digital circuits performing these operations are combinational circuits. These circuits may be designed using the design procedure for combinational logic presented in Chapter 6. This classical procedure assumes that, given two circuits that perform the same function, the one that requires fewer gates is preferable because it will cost less. This is not necessarily true when integrated circuits are used. Since several logic gates are included in a single IC package, it becomes economical to use as many of the gates from an already used package even if, by doing so, we increase the total number of gates. Moreover, some of the interconnections among gates in many ICs are internal to the chip and it is more economical to use as many internal interconnections as possible in order to minimize the number of wires between external pins. With integrated circuits, it is not the count of gates that determines the cost but the number and type of ICs employed and the number of external interconnections needed to implement the given function.

There are numerous occasions where the classical method of Chapter 6 will not produce the best combinational circuit for implementing a given function. Moreover, the truth table and the simplification procedure in this method become too cumbersome if the number of input variables is excessively large. The final circuit obtained dictates that it be implemented with a random connection of SSI gates, which may require a relatively large number of ICs and interconnecting wires. In many cases the application of an alternate design procedure can produce a combinational circuit for a given function which is far better than the one obtained by following the classical design method. The possibility of an alternate design procedure depends on the particular problem and the ingenuity of the designer. The classical method constitutes a general procedure that, if followed, guarantees to produce a result. However, before applying the classical method, it is always wise to investigate the possibility of an alternate method that may be more efficient for the particular problem at hand.

The first question that must be answered before going through a detailed design of a combinational circuit is whether the function is already available in an IC package. Numerous MSI devices are available commercially. These devices perform specific digital functions commonly employed in the design of digital systems. If an MSI device cannot be found to produce exactly the function needed, a resourceful designer may be able to formulate a method so as to incorporate an MSI device in this circuit. The selection of MSI components in preference to SSI gates is extremely important, since it would invariably result in a considerable reduction of IC packages and interconnecting wires.

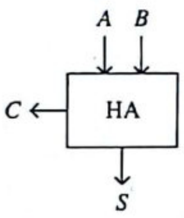
In this chapter, the basic functions are designed using classical design method and some are designed using nonclassical design method. The more complicated functions are designed by incorporating MSI devices. After going through the design examples, a reader will be familiar with when to use classical method and when to uses nonclassical method.

7.2 Adders

The most basic arithmetic operation is the addition of two binary digits. This simple addition consists of four possible elementary operations, namely, 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, and 1 + 1 = 10. The first three operations produce a sum whose length is one bit, but when both augend and addend bits are equal to 1, the binary sum consists of two bits. The higher significant bit of this result is called a *carry*. When the augend and addend numbers contain more significant bits, the carry obtained from the addition of two is added to the next higher-order pair of significant bits. A combinational circuit that performs the addition of two bits is called a *half-adder*. One that performs the addition of three bits (two significant bits and a previous carry) is a *full-adder*. The name of the former stems from the fact that two half-adders can be employed to implement a full-adder. The design of two adder circuits using classical method is presented next.

7.2.1 Half-Adder

From the verbal explanation of a half-adder, we find that this circuit needs two binary inputs and two binary outputs. The input variables designate the augend and addend bits; the output variables produce the sum and carry. We arbitrarily assign A and B to the two inputs and S (for sum) and C (for carry) to the outputs. The block diagram and the truth table of the half-adder are shown in Figure 7.1.



(a) Block diagram

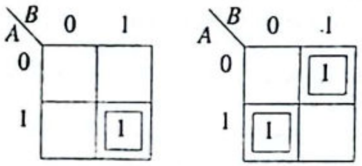
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

(b) Truth table

Figure 7.1 Block diagram and truth table of the half-adder.

The K-map simplification of the output functions of the half-adder is shown in Figure 7.2, which gives

$$S = A'B + AB' = A \oplus B$$
$$C = AB$$



(a) $C = AB$ (b) $S = A'B + AB'$

Figure 7.2 K-map simplification of output functions of the half-adder.

The implementation of the half-adder using AND-OR gates is shown in Figure 7.3(a) and using EXOR gate is shown in Figure 7.3(b).

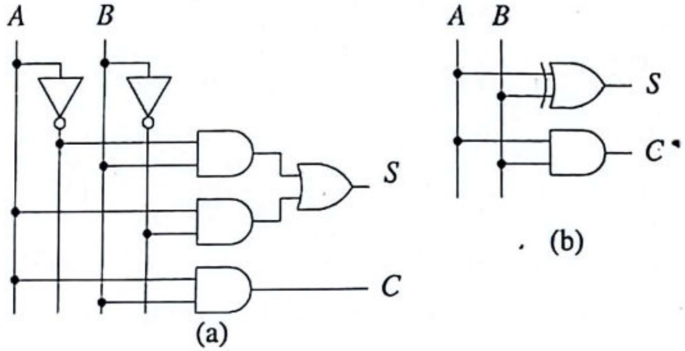
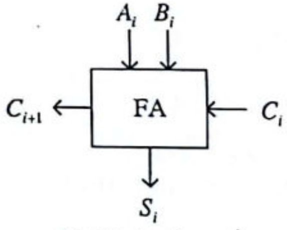


Figure 7.3 Implementation of half-adder.

7.2.2 Full-Adder

A full-adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of three inputs and two outputs. The block diagram of the full-adder is shown in Figure 7.4(a). Two of the input variables, denoted by A_i and B_i, represent the two significant bits to be added. The third input, C_i, represents the carry input from the previous lower significant position. The two outputs are designated by the symbols S_i for sum and C_{i+1} for carry output to the next higher significant position. The truth table of the full-adder is shown in Figure 7.4(b).



(a) Block diagram

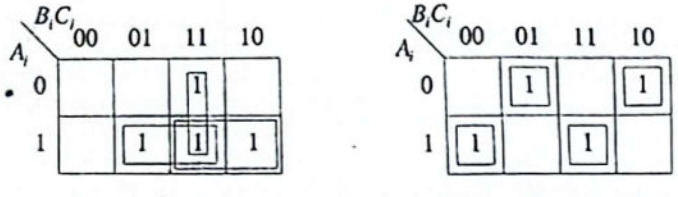
A_i	B_i	C_i	C_{i+1}	S_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(b) Truth table

Figure 7.4 Block diagram and truth table for the full-adder.

The K-map simplification of the output functions of the full-adder is shown in Figure 7.5, which gives

$$S_i = A_i'B_i'C_i + A_i'B_iC_i' + A_iB_i'C_i' + A_iB_iC_i$$
$$C_{i+1} = A_iB_i + A_iC_i + B_iC_i$$



(a) $C_{i+1} = A_iB_i + A_iC_i + B_iC_i$ (b) $S_i = A_i'B_i'C_i + A_i'B_iC_i' + A_iB_i'C_i' + A_iB_iC_i$

Figure 7.5 K-map simplification of output functions of the full-adder.

The logic diagram for the full-adder implementation is shown in Figure 7.6.

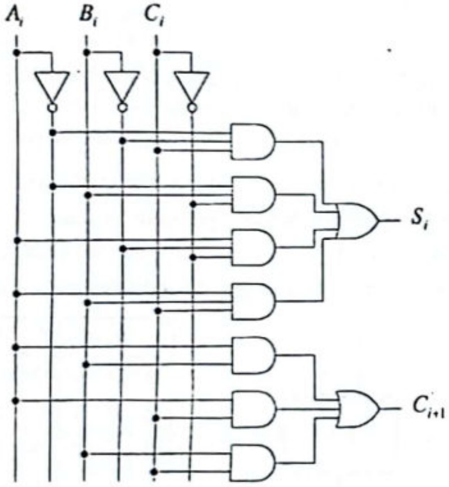
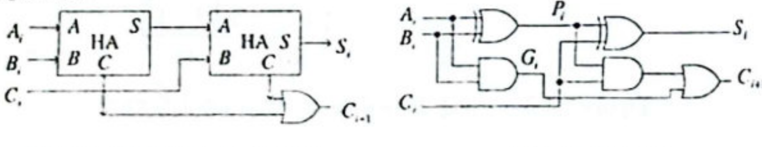


Figure 7.6 Implementation of the full-adder.

7.2.3 Full-Adder using Half-Adder

A full-adder can be implemented with two half-adders and one OR gate, as shown in Figure 7.7.



(a) Connection using block diagram (b) Logic diagram

Figure 7.7 Implementation of a full-adder with two half-adders and an OR gate.

The output functions can be verified as follows:

$$P_i = A_i \oplus B_i$$
$$G_i = A_iB_i$$

The expression for S_i is

$$\begin{aligned} S_i &= P_i \oplus C_i \\ &= P_iC_i' + P_i'C_i \\ &= (A_i \oplus B_i)C_i' + (A_i \oplus B_i)'C_i \\ &= (A_iB_i' + A_i'B_i)C_i' + (A_iB_i + A_i'B_i')C_i \\ &= A_iB_i'C_i' + A_i'B_iC_i' + A_iB_iC_i + A_i'B_i'C_i \end{aligned}$$

The expression for C_{i+1} is

$$\begin{aligned} C_{i+1} &= G_i + P_iC_i \\ &= A_iB_i + (A_i \oplus B_i)C_i \\ &= A_iB_i + (A_iB_i' + A_i'B_i)C_i \\ &= A_iB_i + A_iB_i'C_i + A_i'B_iC_i \\ &= A_iB_i(C_i + C_i') + A_iB_i'C_i + A_i'B_iC_i \\ &= (A_iB_iC_i + A_iB_iC_i') + (A_iB_iC_i + A_iB_i'C_i) + (A_iB_iC_i + A_i'B_iC_i) \\ &= A_iB_i + A_iC_i + B_iC_i \end{aligned}$$

These expressions verify the correctness of the full-adder implementation of Figure 7.7.

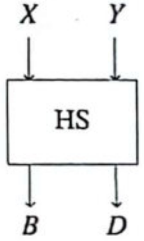
7.3 Subtractor

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend (Section 2.5). By this method, the subtraction operation becomes an addition operation requiring full-adder for its machine implementation. It is, however, possible to implement subtraction with logic circuits in a direct manner, as done with paper and pencil. By this method, each subtrahend bit of the number is subtracted from its corresponding significant minuend bit to form a difference bit. If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position. The fact that a 1 has been borrowed must be conveyed to the next higher pair of bits by means of a binary signal coming out (output) of a given stage and going into (input) the next higher stage. Just as there are half- and full-adders, there are half- and full-subtractors.

54 27 27

7.3.1 Half-Subtractor

A half-subtractor is a combinational circuit that subtracts two bits and produces their difference. It also has an output to specify if a 1 has been borrowed. The block diagram of the half-subtractor is shown in Figure 7.8(a). We arbitrarily designate the minuend bit by X and the subtrahend bit by Y. To perform $X - Y$, we have to check the relative magnitudes of X and Y. If $X \geq Y$, we have three possibilities: 0 – 0 = 0, 1 – 0 = 1, and 1 – 1 = 0. The results is called the *difference bit*. If $X < Y$, we have 0 – 1, and it is necessary to borrow a 1 from the next higher stage. The 1 borrowed from the next higher stage adds 2 to the minuend bit. With the minuend equal to 2, the difference becomes 2 – 1 = 1. The half-subtractor needs two outputs. One output generates the difference and will be designated by the symbol D. The second output, designated B for borrow, generates the binary signal that informs the next stage that a 1 has been borrowed. The truth table of the half-subtractor is shown in Figure 7.8(b).



(a) Block diagram

X	Y	B	D
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

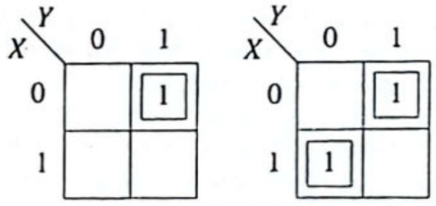
(b) Truth table

Figure 7.8 Block diagram and truth table of the half-subtractor.

The K-map simplification of the output functions of the half-subtractor is shown in Figure 7.9, which gives

$$D = X'Y + XY' = X \oplus Y$$

$$B = X'Y$$



(a) $B = X'Y$ (b) $D = X'Y + XY'$

Figure 7.9 K-map simplification of the output functions of the half-subtractor.

The implementation of the half-subtractor is shown in Figure 7.10.

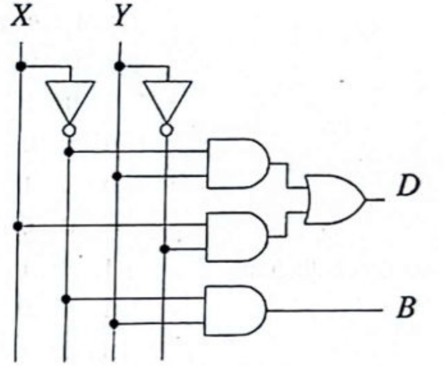
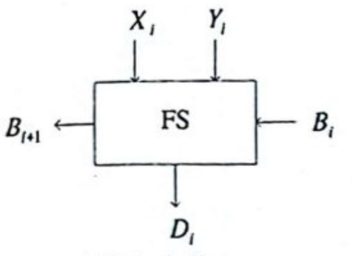


Figure 7.10 Implementation of the half-subtractor.

It is interesting to note that the logic for D is exactly the same as the logic for S output in the half-adder.

7.3.2 Full-Subtractor

A full-subtractor is a combinational circuit that performs a subtraction between two bits, taking into account that a 1 may have been borrowed by a lower significant stage. This circuit has three inputs and two outputs. The three inputs, X_i, Y_i, and B_i denote the minuend, subtrahend, and previous borrow, respectively. The two outputs, D_i and B_{i+1}, represent the difference and output borrow, respectively. The block diagram of the full-subtractor is shown in Figure 7.11(a) and its truth table is shown in Figure 7.11(b). The values of the output variables are determined from the subtraction of $X_i - Y_i - B_i$. The input combinations having input borrow $B_i = 0$ reduce to the same four conditions of the half-subtractor. For $X_i = 0$, $Y_i = 0$, and $B_i = 1$, we have to borrow a 1 from the next stage, which makes $B_{i+1} = 1$ and adds 2 to X_i. Since 2 – 0 – 1 = 1, $D_i = 1$. For $X_i = 0$, $Y_i = 1$, and $B_i = 1$, we need to borrow again, making $B_{i+1} = 1$ and $X_i = 2$. Since $2 - 1 - 1 = 0$, $D_i = 0$. For $X_i = 1$, $Y_i = 0$, and $B_i = 1$, we have $X_i - Y_i - B_i = 0$, which makes $B_{i+1} = 0$ and $D_i = 0$. Finally, for $X_i = 1$, $Y_i = 1$, and $B_i = 1$, we have to borrow 1, making $B_{i+1} = 1$ and $X_i = 3$, and 3 – 1 – 1 = 1, making $D_i = 1$.



(a) Block diagram

X_i	Y_i	B_i	B_{i+1}	D_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

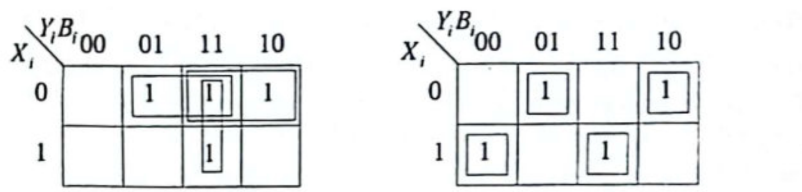
(b) Truth table

Figure 7.11 Block diagram and truth table of the full-subtractor.

The K-map simplification of the output functions is shown in Figure 7.12, which gives

$$D_i = X_i'Y_i'B_i + X_i'Y_iB_i' + X_iY_i'B_i' + X_iY_iB_i$$

$$B_{i+1} = X_i'Y_i + X_i'B_i + Y_iB_i$$



(a) $B_{i+1} = X_i'Y_i + X_i'B_i + Y_iB_i$ (b) $D_i = X_i'Y_i'B_i + X_i'Y_iB_i' + X_iY_i'B_i' + X_iY_iB_i$

Figure 9.12 K-map simplification of the full-subtractor.

Again we note that the logic function for output D_i in the full-subtractor is exactly the same as output S_i in the full-adder. Moreover, the output B_{i+1} resembles the function for C_{i+1} in the full-adder, except that the input variable X_i is complemented. Because of these similarities, it is possible to convert a full-adder into a full-subtractor by merely complementing X_i prior to its application to the gates that form the carry output. The implementation of the full-subtractor is shown in Figure 7.13.

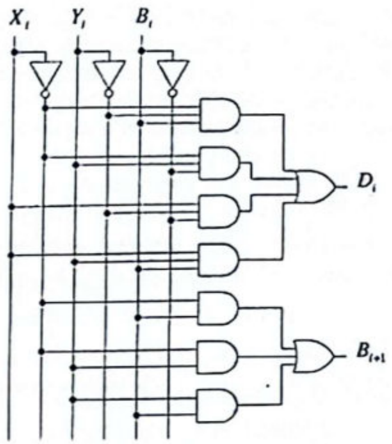


Figure 7.13 Implementation of the full-subtractor.

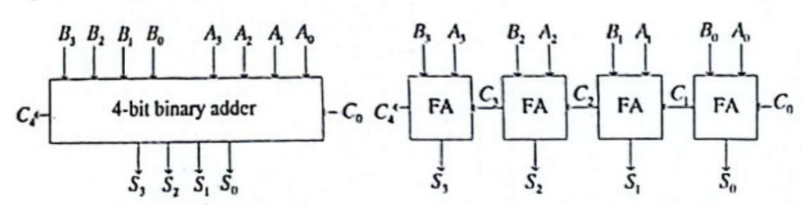
7.4 Binary Parallel Adder

The full-adder introduced in Section 7.2 forms the sum of two bits and a previous carry. Two binary numbers of *n* bits each can be added by means of this circuit. To demonstrate with a specific example, consider two binary numbers, *A* = 1010 and *B* = 0011, whose sum is *S* = 1110. When a pair of bits are added through a full-adder, the circuit produces a carry to be used with the pair of bits one significant position higher. This is shown below.

Subscript	3 2 1 0
Augend, A_i	1 0 1 1
Addend, B_i	0 0 1 1
Input Carry, C_i	0 1 1 0
Sum, S_i	1 1 1 0
Output Carry, C_{i+1}	0 0 1 1

The bits are added with full-adders, starting from the least significant position (subscript 0), to form the sum bit and carry bit. The input carry C_0 in the least significant position must be 0. The value of C_{i+1} is the output carry of the full-adder in the *i*th significant position. This value is transferred into the input carry of the full-adder that adds the bits in the ($i+1$)st significant position, that is, one higher significant position to the left. The sum bits are thus generated starting from the rightmost position and are available as soon as the corresponding previous carry bit is generated.

A *binary parallel adder* (or *binary adder* in short) is a digital function that produces the arithmetic sum of two binary numbers in parallel. It consists of full-adders connected in cascade, with the output carry from one full-adder connected to the input carry of the next full-adder. The block diagram of a 4-bit binary adder is shown in Figure 7.14(a) and its implementation using four full-adders is shown in Figure 7.14(b).



(a) Block diagram (b) Implementation using full-adders

Figure 7.14 4-bit binary adder.

All bits of A and B are applied simultaneously. The output carry from one full-adder is connected to the input carry of the full-adder one position to its left. As soon as the carries are generated, the correct sum bits emerge from the sum outputs of all full-adders.

An n-bit binary adder requires n full-adders. It can be constructed from 4-bit, 2-bit, and 1-bit binary adder ICs by cascading several packages. The output carry from one package must be connected to the input carry of the one with the next higher-order bits. Cascading of two 4-bit binary adders to implement a 8-bit binary adder is shown in Figure 7.15. The output carry of the leftmost stage indicates the overflow, that is, a 1 in C_8 indicates that the sum exceeds 11111111.

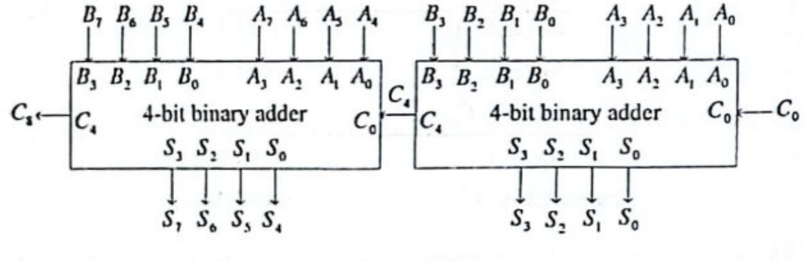


Figure 7.15 Cascading of two 4-bit binary adders to implement a 8-bit binary adder.

The 4-bit binary adder is a typical example of an MSI function. It can be used in many applications involving arithmetic operations. Observe that the design of this circuit by the classical method would require a truth table with $2^9 = 512$ entries, since there are nine inputs to the circuit. By using an iterative method of cascading an already known function, we were able to obtain a simple and well-organized implementation.

7.5 Binary Adders with Look-Ahead Carry

The addition of two binary numbers in parallel implies that all the bits of the augend and the addend are available for computation at the same time. As in any combinational circuit, the signal must propagate through the gates before the correct output sum is available in the output terminals. The total propagation time is equal to the propagation delay of a typical gate times the number of gate levels in the circuit. The longest propagation delay time in a parallel adder is the time it takes the carry to propagate through the full-adders. Since each bit of the sum output depends on the value of the input carry, the value of S_i in any given stage in the adder will be in its steady-state final value only after the input carry to that stage has been propagated. Consider output S_3 in Figure 7.14(b). Inputs A_3 and B_3 reach a steady value as soon as input signals are applied to the adder. But input carry C_3 does not settle to its final steady-state value until C_2 is available in its steady-state value. Similarly, C_2 has to wait for C_1, and so on down to C_0. Thus only after the carry propagates through all stages, the last output S_3 and carry C_4 will settle to their final steady-state value.

The number of gate levels for the carry propagation can be found from the circuit of the full-adder of Figure 7.7(b). The signals at P_i and G_i settle to their steady-state value after the propagation through their respective gates. These two signals are common to all full-adders and depend only on the input augend and addend bits. The signal from the input carry, C_i, to the output carry, C_{i+1}, propagates through an AND gate and an OR gate, which constitutes two gate levels. If there are four full-adders in the parallel adder, the output carry C_4 would have $2 \times 4 = 8$ gate levels from C_0 to C_4. The total propagation time in the adder would be the propagation time in one half-adder plus eight gate levels. For an n-bit parallel adder, there are $2n$ gate levels for the carry to propagate through.

The carry propagation time is a limiting factor on the speed with which two numbers are added in parallel. Although a parallel adder, or any combinational circuit, will always have some value at its output terminals, the outputs will not be correct unless the signals are given enough time to propagate through the gates connected from the inputs to the outputs. Since all other arithmetic operations are implemented by successive additions, the time consumed during the addition process is very critical. An obvious solution for reducing the carry propagation delay times is to employ faster gates with reduced delays. But physical circuits have a limit to their capacity. Another solution is to increase the equipment complexity in such a way that the carry delay time is reduced. There are several techniques for reducing the carry propagation time in a parallel adder. The most widely used technique employs the principle of *look-ahead* carry.

Consider the circuit of the full-adder shown in Figure 7.7(b). The output sum and carry can be expressed as

$$S_i = P_i \oplus C_i$$
$$C_{i+1} = G_i + P_iC_i$$

where

$$P_i = A_i \oplus B_i$$
$$G_i = A_iB_i$$

G_i is called a *carry generate* and it produces an output carry when both A_i and B_i are one, regardless of the input carry. P_i is called a carry propagate because it is the term associated with the propagation of the carry from C_i to C_{i+1}.

We now write the Boolean function for the carry output of each stage and substitute for each C_i its value from the previous equations.

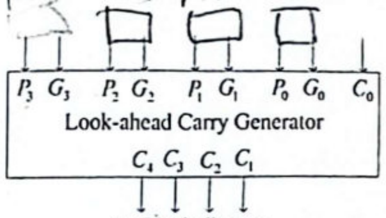
$$C_1 = G_0 + P_0C_0$$

$$\begin{aligned} C_2 &= G_1 + P_1C_1 \\ &= G_1 + P_1(G_0 + P_0C_0) \\ &= G_1 + P_1G_0 + P_1P_0C_0 \end{aligned}$$

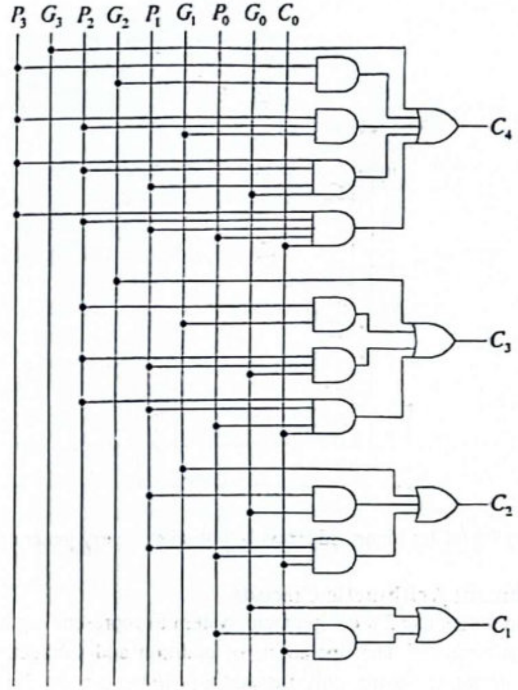
$$\begin{aligned} C_3 &= G_2 + P_2C_2 \\ &= G_2 + P_2(G_1 + P_1G_0 + P_1P_0C_0) \\ &= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0 \end{aligned}$$

$$\begin{aligned} C_4 &= G_3 + P_3C_3 \\ &= G_3 + P_3(G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0) \\ &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0. \end{aligned}$$

Since the Boolean functions for each output carry is expressed in sum of products form, each function can be implemented with one level of AND gates followed by an OR gate. The circuit implementing these functions is called *look-ahead carry generator*. The block diagram and logic implementation of 4-bit look-ahead carry generator is shown in Figure 7.16. Note that C_4 does not have to wait for C_3, C_2, and C_1 to propagate; in fact, C_4 is propagated at the same time as C_3, C_2, and C_1.



(a) Block diagram



(b) Logic diagram

Figure 7.16 4-bit look-ahead carry generator.

The construction of a 4-bit binary adder with a look-ahead carry generator is shown in Figure 7.17. Each sum output requires two EXOR gates. The output of the first EXOR gate generates the P_i variable, and the AND gate generates the G_i variable. All the *P*s and *G*s are generated in one gate level. The carries are propagated through the look-ahead carry generator and applied as inputs to the second EXOR gate. After the *P* and *G* signals settle into their steady-state values, all output carries are generated after a delay of two levels of gates. Thus, outputs S_1 through S_3 have equal propagation delay times.

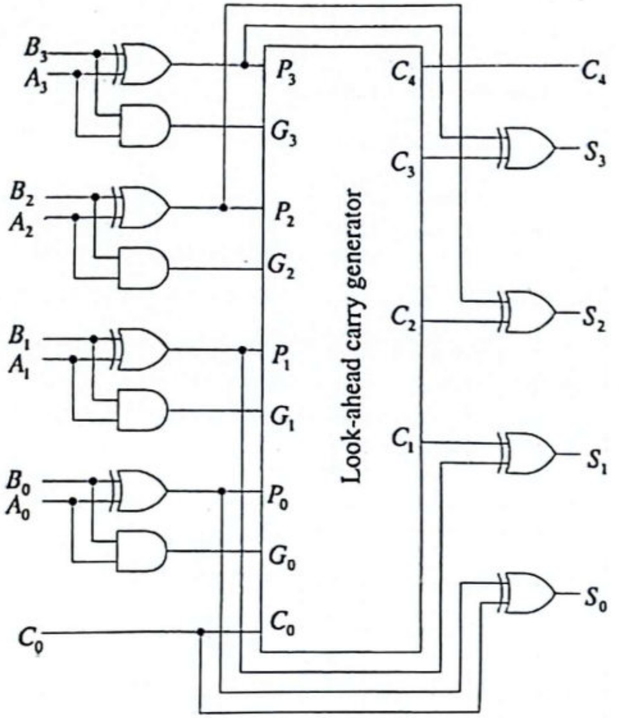

B_3
A_3
B_2
A_2
B_1
A_1
B_0
A_0
C_0
P_3
G_3
P_2
G_2
P_1
G_1
P_0
G_0
C_0
Look-ahead carry generator
C_4
C_3
C_2
C_1
C_4
S_3
S_2
S_1
S_0

Figure 7.17 4-bit binary adder with look-ahead carry generator.

7.6 2's Complement Arithmetic Circuits

Most modern computers use 2's complement system to represent negative numbers and to perform subtraction. The operations of addition and subtraction of signed numbers can be performed using only the addition if we use the 2's complement form to represent negative numbers.

7.6.1 2's Complement Adder

Positive and negative numbers, including the sign bits, can be added together in the basic binary adder circuit of Figure 7.14 when the negative numbers are in 2's complement form. This is illustrated in Figure 7.18 for the addition of -3 and $+6$. The -3 is represented in its 2's complement form as 1101, where the first 1 is the sign bit; the $+6$ is represented in its 2's complement form as 0110, with the first 0 as the sign bit. The 4-bit binary adder produces sum outputs of 0011, which represent $+3$ in 2's complement form. The C_4 output is 1 but is disregarded in the 2's complement method.

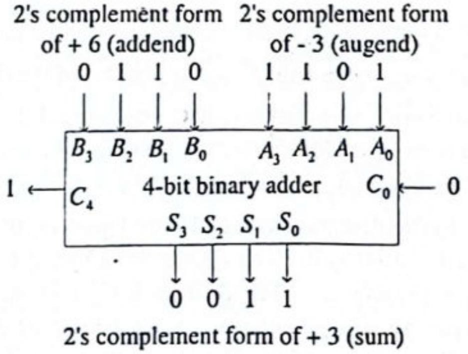

2's complement form of + 6 (addend)
2's complement form of - 3 (augend)
0 1 1 0
1 1 0 1
B_3 B_2 B_1 B_0
A_3 A_2 A_1 A_0
1
C_4
4-bit binary adder
C_0
0
S_3 S_2 S_1 S_0
0 0 1 1
2's complement form of + 3 (sum)

Figure 7.18 Addition of negative and positive number in 2's complement system using binary adder.

7.6.2 2's Complement Subtractor

When the 2's complement system is used, the subtrahend is changed to its 2's complement and then added to the minuend. The sum then represents the difference between the minuend and subtrahend.

The binary adder can be adapted to perform the subtraction of 2's complement signed number if we provide a means for taking the 2's complement of the subtrahend B. The 2's complement of a binary number is obtained by complementing each bit and then adding 1 to the LSB. Figure 7.19 shows how this can be accomplished. The EXOR gates connected to the inputs B_3 through B_0 of the binary adder produce the complement of each bit of subtrahend B. The value of C_0 is 1, so it adds an extra 1 into the LSB of the adder; this forms the 2's complement of B.

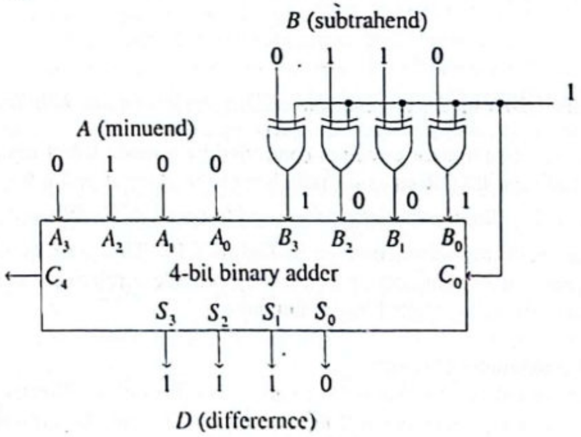

B (subtrahend)
0 1 1 0
1
A (minuend)
0 1 0 0
1 0 0 1
A_3 A_2 A_1 A_0
B_3 B_2 B_1 B_0
0
C_4
4-bit binary adder
C_0
S_3 S_2 S_1 S_0
1 1 1 0
D (difference)

Figure 7.19 4-bit 2's complement subtractor using 4-bit binary adder.

The outputs S_3 through S_0 represent the results of the subtraction operation in 2's complement form. Of course S_3 is the sign bit of the result and indicates whether the result is + or –. The carry output C_4 is disregarded.

To help clarify this operation, subtracting + 6 from + 4 is shown in Figure 7.19. + 4 is applied to the inputs A_3 through A_0 as 0100. + 6 is applied as 0110, which is complemented by EXOR gates and applied to the inputs B_3 through B_0 as 1001. The 1001 is added to 0100 by the binary adder along with a 1 added to the LSB position. This produces sum output bits 1110 and a $C_4 = 0$, which is disregarded. This 1110 represents the required difference. Since the sign bit is 1, it is a negative number and is in 2's complement form. The 1110 represents -2 in 2's complement system.

An n-bit 2's complement subtractor can be implemented with n-bit binary adder in a similar manner.

7.6.3 Combined 2's Complement Adder/Subtractor

It should now be clear that the basic binary adder can be used to perform addition or subtraction depending on whether the B number is left unchanged or is converted to 2's complement. A complete circuit that can perform both addition and subtraction in the 2's complement system is shown in Figure 7.20.

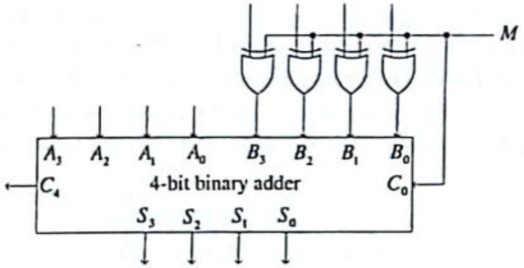


Figure 7.20 4-bit 2's complement adder/subtractor using 4-bit binary adder.

This adder/subtractor circuit is controlled by a mode select input M. When the M input is 0, the EXOR gates do not change the B input and a 0 is applied to the carry input C_0. So, the circuit performs addition $A + B$. When the M input is 1, the circuit performs subtraction as in Figure 7.19. This type of adder/subtractor circuits are used in computers because they provide a relatively simple means for adding and subtracting signed binary numbers.

7.7 BCD Arithmetic Circuits

Computers or calculators that perform arithmetic operations directly in the decimal number system represent decimal numbers in BCD form. An arithmetic circuit for such a computer must accept BCD numbers and present results in the BCD number. For binary arithmetic, it was sufficient to consider a pair of significant bits at a time, together with a previous carry or borrow. But a BCD arithmetic circuit requires nine inputs and five outputs, since four bits are required to code each decimal digit and the circuit must have an input carry or borrow and an output carry or borrow. The design of a nine-input, five-output combinational circuit by the classical method requires a truth table with $2^9 = 512$ entries. Many of the input combinations are don't care conditions, since BCD code input has six combinations that are invalid. The simplified Boolean functions for the circuit may be obtained by a computer-generated Quine-McCluskey method, and the result would probably be a connection of gates forming an irregular pattern. An alternate procedure is to add or subtract the numbers with binary adder circuits, taking into consideration the fact that six combinations in each 4-bit input are not used. The output must be modified so that only those binary combinations that are valid combinations of the BCD are generated.

7.7.1 BCD Adder

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input carry. Suppose we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in *binary* and produce a result that may range from 0 to 19. These binary numbers are listed in Table 7.1 and are labeled by symbols K, Z_3, Z_2, Z_1, and Z_0. K is the carry; Z_3, Z_2, Z_1, and Z_0 are sum bits. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit *binary* adder. The output sum of two *decimal digits* must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary number in the first column can be converted to the correct BCD-digit representation of the number in the second column.

In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain an invalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry $K = 1$. The other six combinations from 1010 to 1111 that need a correction have a 1 in position Z_3. To distinguish them from binary 1000 and 1001, which also have a 1 in position Z_3, we specify further that either Z_2 or Z_1 must have a 1. The condition for a correction and an output carry can be expressed by the Boolean function

$$C = K + Z_3Z_2 + Z_3Z_1$$

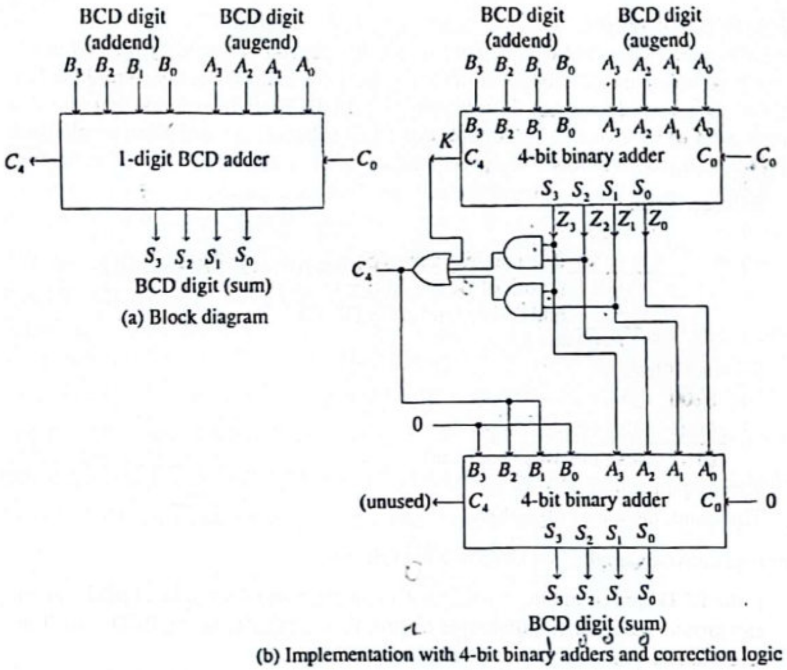
When $C = 1$, it is necessary to add 0110 to the binary sum and provide an output carry for the next stage.

Table 7.1 Derivation of BCD adder.

Binary sum	BCD sum	Decimal
K $Z_3 Z_2 Z_1 Z_0$	C $S_3 S_2 S_1 S_0$	
0 0 0 0 0	0 0 0 0 0	0
0 0 0 0 1	0 0 0 0 1	1
0 0 0 1 0	0 0 0 1 0	2
0 0 0 1 1	0 0 0 1 1	3
0 0 1 0 0	0 0 1 0 0	4
0 0 1 0 1	0 0 1 0 1	5
0 0 1 1 0	0 0 1 1 0	6
0 0 1 1 1	0 0 1 1 1	7
0 1 0 0 0	0 1 0 0 0	8
0 1 0 0 1	0 1 0 0 1	9
0 1 0 1 0	1 0 0 0 0	10
0 1 0 1 1	1 0 0 0 1	11
0 1 1 0 0	1 0 0 1 0	12
0 1 1 0 1	1 0 0 1 1	13
0 1 1 1 0	1 0 1 0 0	14
0 1 1 1 1	1 0 1 0 1	15
1 0 0 0 0	1 0 1 1 0	16
1 0 0 0 1	1 0 1 1 1	17
1 0 0 1 0	1 1 0 0 0	18
1 0 0 1 1	1 1 0 0 1	19

A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also BCD. The block diagram of a BCD adder is shown in Figure 7.21(a). A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Figure 7.21(b). The two decimal digits, together with the input carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder. The output carry generated from the bottom binary adder can be ignored, since it supplies information already available at the output-carry terminal.

8 + 9
1000 1001
5 + 3
0101 + 0011


BCD digit (addend)
BCD digit (augend)
B_3 B_2 B_1 B_0 A_3 A_2 A_1 A_0
C_4
1-digit BCD adder
C_0
S_3 S_2 S_1 S_0
BCD digit (sum)
K
C_4
4-bit binary adder
C_0
Z_3 Z_2 Z_1 Z_0
0
(unused)
0

(a) Block diagram

(b) Implementation with 4-bit binary adders and correction logic

Figure 7.21 1-digit BCD adder.

An n-digit BCD adder needs n cascaded BCD adder stages. The output carry from one stage must be connected to the input carry of the next higher-order stage. Cascading of two BCD adders is shown in Figure 7.22.

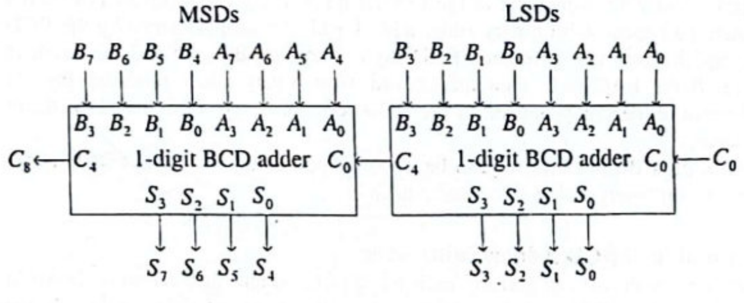

MSDs
LSDs
B_7 B_6 B_5 B_4 A_7 A_6 A_5 A_4
B_3 B_2 B_1 B_0 A_3 A_2 A_1 A_0
C_8
C_4
1-digit BCD adder
C_0
S_3 S_2 S_1 S_0
S_7 S_6 S_5 S_4

Figure 7.22 Cascading of two 1-digit BCD adders to implement a 2-digit BCD adder.

7.7.2 BCD Subtractor

For BCD subtraction, 9's complement of the subtrahend is added to the minuend. The 9's complement of a BCD number is given by nine minus that number. For example, 9's complement of 7 is 9 − 7 = 2. In BCD code we say that the 9's complement of 0111 is 0010. Examples of BCD subtraction using 9's complement to represent negative numbers are given below:

(a) Subtract 5 from 9.
9 = 1001
−5 = 0100 (9's complement of 5)
10011 (result of BCD addition)
1 add end around carry (EAC)
0100 = + 4

(b) Subtract 8 from 4
4 = 0100
−8 = 0001 (9's complement of 8)
0101 (result of BCD addition)
9's complement of 0101 or 5 is 4
Therefore, the answer is − 4

From the above examples, we conclude the following:

1. If the BCD sum of minuend and the 9's complement of subtrahend produces an end around carry (EAC) (example of (a)), then add EAC to the BCD sum. The result is positive number represented by this sum.
2. If the BCD sum of the minuend and the 9's complement of the subtrahend does not produce any EAC, then the result is negative and is in the 9's complement form (example of (b)).

The 9's complement of a BCD digit can be determined by adding 1010 to the 1's complement of the digit. Figure 7.23 shows a 1-digit BCD subtractor. The top 4-bit binary adder produces 9's complement of the subtrahend *B* by complementing the bits by EXOR gates and then adding 1010. The BCD adder of the middle add the minuend *A* and 9's complement of the subtrahend *B*. If the end around carry (EAC) is 1, then the bottom 4-bit binary adder adds 1 with the sum produced by the BCD adder, which gives the difference BCD digit. The sign bit is 0 and the result is positive. If the EAC is 0, then the bottom 4-bit binary adder produces the 9's complement of the sum produced by the BCD adder. The sign bit is 1 and the result is negative.

An *n*-digit BCD subtractor can be implemented using one *n*-digit BCD adder and two 4*n*-bit binary adders in a similar manner.

7.7.3 Combined BCD Adder/Subtractor

A complete circuit that can perform both BCD addition and subtraction is shown in Figure 7.24. This BCD adder/subtractor circuit is controlled by a mode selector input *M*. When the *M* input is 0, the top 4-bit binary adder adds 0 with *B*, that is, it passes *B* unchanged. The middle BCD adder add *A* and *B*. The bottom 4-bit adder adds 0 with the sum produced by the middle BCD adder. Therefore, the total circuit produces $A + B$. When the *M* input is 1, the total circuit produces $A - B$ as in Figure 7.23.

We observe that the BCD arithmetic is rather complex in comparison to the straight binary arithmetic. Therefore, BCD arithmetic requires more hardware and results in reduction of speed. Also, more number of bits is required to represent a given number in BCD. Therefore, BCD is normally not used in computers. However, digital calculators use BCD because the input data from the keyboard as well as the output display are decimal.

7.8 Available MSI Arithmetic ICs

7.8.1 Look-Ahead Carry Generator

A typical look-ahead carry generator is the IC type 74182. The pin assignment of the IC is shown in Figure 7.25. The IC 74182 accepts four active-LOW Carry Propagate (P_0', P_1', P_2', P_3') signals, four active-LOW Carry Generate (G_0', G_1', G_2', G_3') signals, and an active-HIGH Carry Input (C_0). It provides anticipated active-HIGH carries (C_1, C_2, C_3). It also provides active-LOW Carry Propagate (P') and Carry Generate (G') outputs, which may be used for generating C_4.

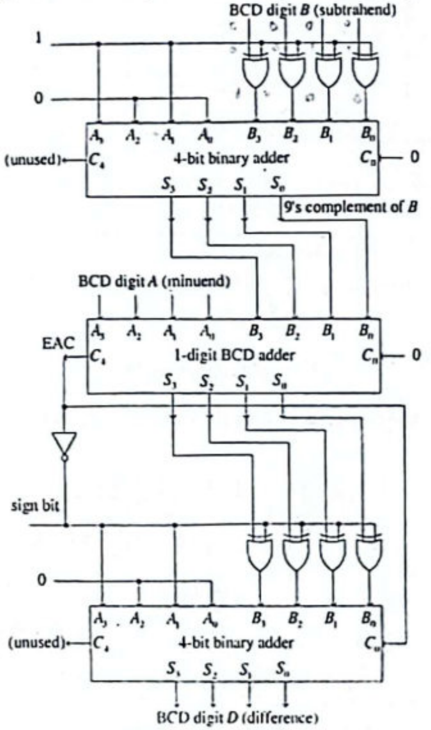


Figure 7.23 1-digit BCD subtractor.

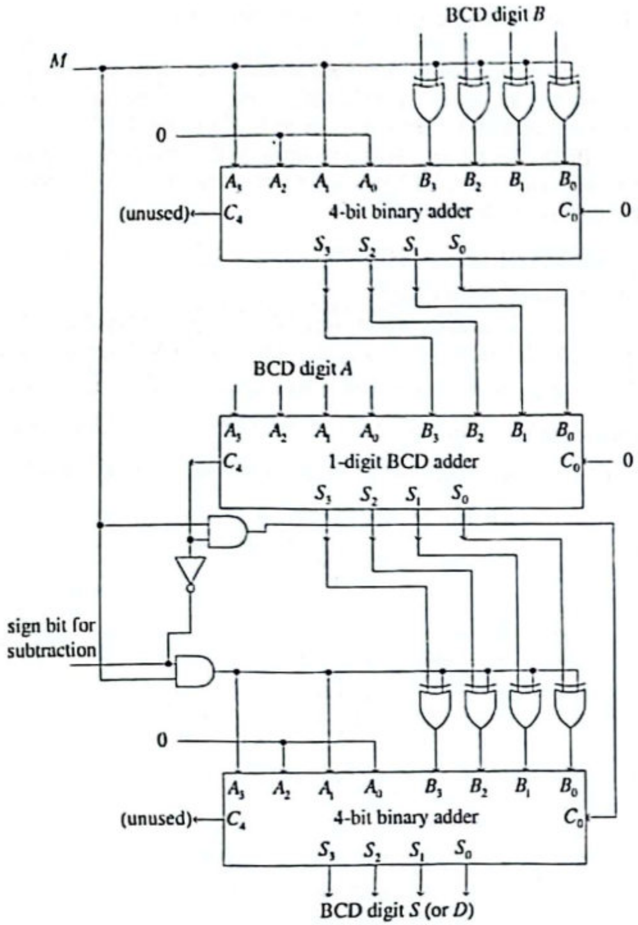

BCD digit B
M
0
A_3 A_2 A_1 A_0 B_3 B_2 B_1 B_0
(unused)
C_4
4-bit binary adder
C_0
0
S_3 S_2 S_1 S_0
BCD digit A
A_3 A_2 A_1 A_0 B_3 B_2 B_1 B_0
C_4
1-digit BCD adder
C_0
0
S_3 S_2 S_1 S_0
sign bit for subtraction
0
A_3 A_2 A_1 A_0 B_3 B_2 B_1 B_0
(unused)
C_4
4-bit binary adder
C_0
S_3 S_2 S_1 S_0
BCD digit S (or D)

Figure 7.24 1-digit BCD adder/subtractor.

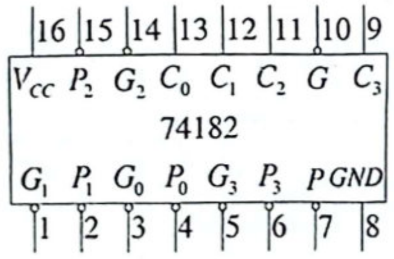

16 15 14 13 12 11 10 9
V_{CC} P_2 G_2 C_0 C_1 C_2 G C_3
74182
G_1 P_1 G_0 P_0 G_3 P_3 P GND
1 2 3 4 5 6 7 8

Figure 9.25 IC type 74182 look-ahead carry generator.

The logic equations provided at the outputs are

$$C_1 = G_0 + P_0C_0$$
$$C_2 = G_1 + P_1G_0 + P_1P_0C_0$$
$$C_3 = G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$$
$$G' = (G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0)'$$
$$P' = (P_3P_2P_1P_0)'$$

C_4 can be generated as follows

$$\begin{aligned} C_4 &= G + PC_0 \\ &= G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0 \end{aligned}$$

7.8.2 4-bit Binary Adder

4-bit binary adder is available as MSI IC package. Available 4-bit binary adder ICs are given in Table 7.2.

Table 7.2 Available MSI binary adder ICs.

IC No.	Description
7483	4-bit binary adder
74283	4-bit binary adder with fast carry

TTL IC type 7483 is discussed here. The internal construction of IC type 7483 is similar to Figure 7.17. The pin assignment of the IC is shown in Figure 7.26. The two 4-bit input binary numbers are A_0 through A_3 and B_0 through B_3. The 4-bit sum is obtained from S_0 through S_3. C_0 is the input carry and C_4 the output carry.

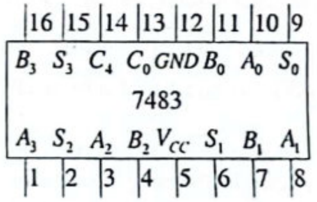

16 15 14 13 12 11 10 9
B_3 S_3 C_4 C_0 GND B_0 A_0 S_0
7483
A_3 S_2 A_2 B_2 V_{CC} S_1 B_1 A_1
1 2 3 4 5 6 7 8

Figure 7.26 IC type 7483 4-bit binary adder.

7.8 Magnitude Comparator

The comparison of two numbers is an operation that determines if one number is greater than, less than, or equal to the other number. A *magnitude comparator* is a combinational circuit that compares two numbers, A and B, and determines their relative magnitudes. The outcome of the comparison is specified by three binary variables that indicate whether $A > B$, $A = B$, or $A < B$.

The circuit for comparing two n-bit numbers has 2^{2n} entries in the truth table and becomes too cumbersome even with $n = 3$. On the other hand, as one may suspect, a comparator circuit possesses a certain amount of regularity. Digital functions which posses an inherent well-defined regularity can usually be designed by means of an algorithmic procedure if one is found to exist. An *algorithm* is a procedure that specifies a finite set of steps that, if followed, gives the solution to a problem. We illustrate this method here by deriving an algorithm for the design of a 4-bit magnitude comparator.

The algorithm is a direct application of the procedure a person uses to compare the relative magnitude of two numbers. Consider two numbers, A and B, with four digits each. Write the coefficients of the numbers with descending significance as follows:

$$A = A_3A_2A_1A_0$$

$$B = B_3B_2B_1B_0$$

where each subscripted letter represent one of the digits in the number. The two numbers are equal if all pairs of significant digits are equal, i.e., if $A_3 = B_3$ and $A_2 = B_2$ and $A_1 = B_1$ and $A_0 = B_0$. When the numbers are binary, the digits are either 1 or 0 and the equality relation of each pair of bits can be expressed logically with an EXNOR or equivalence function:

$$x_i = A_iB_i + A_i'B_i' \quad i = 0,1,2,3$$

where $x_i = 1$ only if the pair of bits in position i are equal, i.e., if both are 1s or both are 0s. The equality of the two numbers, A and B, is displayed in a combinational circuit by an output binary variable which we designate by the symbol $(A = B)$. This binary variable is equal to 1 if the input numbers, A and B, are equal, and it is equal to 0 otherwise. For the equality to exist, all x_i variables must be equal to 1. This dictates an AND operation of all variables:

$$(A = B) = x_3x_2x_1x_0$$

The binary variable $(A = B)$ is equal to 1 only if all pairs of bits of the two numbers are equal.

To determine if A is greater than or less than B, we inspect the relative magnitudes of pairs of significant bits starting from the most significant position. If the two bits are equal, we compare the next lower significant pair of bits. This comparison continues until a pair of unequal bits is reached. If the corresponding bit of A is 1 and that of B is 0, we conclude that $A > B$. If the corresponding bit of A is 0 and that of B is 1, we have that $A < B$. The sequential comparison can be expressed logically by the following two Boolean functions:

$$(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$$

$$(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$$

The symbols $(A > B)$ and $(A < B)$ are *binary* output variables which are equal to 1 when $A > B$ or $A < B$, respectively.

The gate implementation of the three output variables just derived is simpler than it seems because it involves a certain amount of repetition. The "unequal" outputs can use the same gates that are needed to generate the "equal" output. The logic diagram of the 4-bit magnitude comparator is shown in Figure 7.25. This is a multilevel implementation and, as clearly seen, it has a regular pattern. The procedure for obtaining magnitude comparator circuits for binary numbers with more that four bits should be obvious from this example. The same circuit can be used to compare the relative magnitudes of two BCD digits.

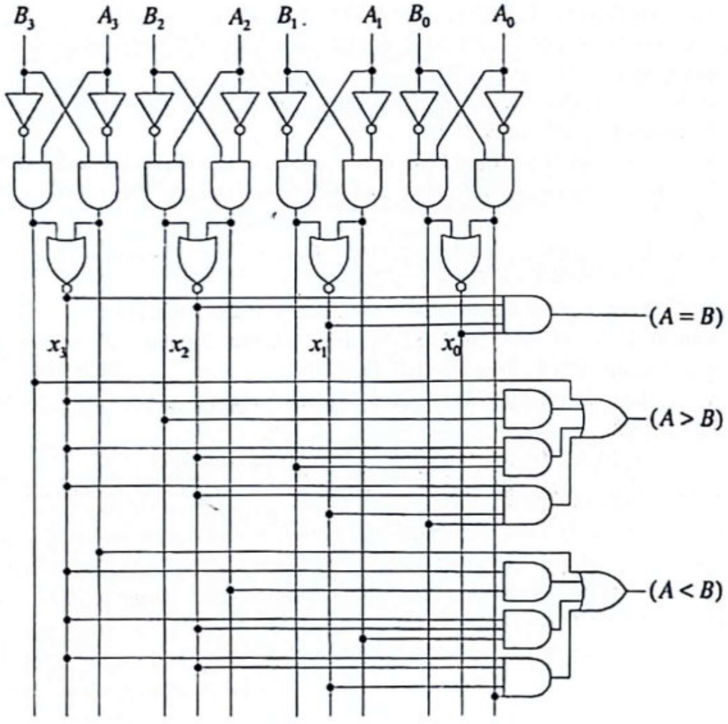


Figure 7.25 4-bit magnitude comparator.

Exercises

7.1 Implement a full-subtractor with two half-subtractors and an OR gate.

7.2 Show how a full-adder can be converted to a full-subtractor with the addition of one NOT gate.

7.3 Design a BCD-to-excess-3 code converter using a 4-bit binary adder.

7.4 Design an excess-3-to-BCD code converted using a 4-bit binary adder.

7.5 Using four 4-bit binary adders, construct a 16-bit binary adder.

7.6 Design a 4-bit BCD adder using 4-bit binary adders.

7.7 Design a circuit that compares two 4-bit numbers, A and B, to check if they are equal. The circuit has one output x, so that $x = 1$ if $A = B$, and $x = 0$ if $A \neq B$.

Further Reading

1. *The TTL Data Book.* Dallas, Texas Instruments, 1988.
2. Richard S. Sandige, *Modern Digital Design*, McGraw-Hill Publishing Company, 1990.
3. R. P. Jain, *Modern Digital Electronics*, 2nd ed., Tata McGraw-Hill Publishing Company Limited, India, 1998.
4. *Microprocessor Data Hand Book*, Revised Ed. BPB Publications, India, 2000.
5. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
6. John P. Uyemura, *Digital Systems Design: An Integrated Approach*, Brooks/Cole Publishing Company, 2000.
7. D. C. Green, *Digital Electronics*, 5th ed., Pearson Education, 2001.
8. Ronald J. Tocci and Neal S. Widmer, *Digital Systems: Principles and Applications*, 8th ed., Prentice-Hall, Inc., 2001.
9. V. K. Jain, *Switching Theory and Digital Electronics*, Khanna Publishers, India, 2002.
10. M. Morris Mano, *Digital Design*, 3rd ed., Pearson Education, 2002.
11. Samuel C. Lee, *Digital Circuits and Logic Design*, Prentice-Hall, Inc., 2002.

CHAPTER 8

ENCODERS, DECODERS, MULTIPLEXERS, AND DEMULTIPLEXERS

8.1 Introduction

Discrete quantities of information are represented in digital systems with binary codes. Sometimes binary codes are used for error detection during digital data transmission. A binary code of n bits is capable of representing up to 2^n distinct elements of the coded information. Discrete inputs to digital systems are encoded to binary-coded form by encoder circuits. In a reverse process, binary-coded data are decoded to discrete information by decoder circuits. Sometimes code conversion between one form to another is needed and is done by special types of encoder called code converter.

Multiplexing means transmitting a large number of information units over a smaller number of channels or lines. Demultiplexing is the reverse process of multiplexing. In demultiplexing process, information received on a smaller number of channels or lines is distributed over a large number of units. Multiplexer and demultiplexer circuits play an important role in digital systems where selection of path between multiple sources and single destination or between a single source and multiple destinations is required.

In this chapter design and application of various encoder, decoder, multiplexer, and demultiplexer circuits are studied.

8.2 Encoders

8.2.1 Design of Encoders

An *encoder* is a combinational circuit that has 2^n (or less) input lines and n output lines. Only one of the input lines is activated at a given time. The output lines generate the binary code corresponding to the activated input. The input lines can be activated in two ways – by setting the activated input to 0 and the other inputs to 1 (active-LOW input) or by setting the activated input to 1 and the other inputs to 0 (active-HIGH input). The block diagram and truth tables of an *octal-to-binary encoder* (also called *8-to-3-line encoder*) for both active-LOW and active-HIGH inputs are shown in Figure 8.1. From the truth tables, the Boolean function for the outputs can be written as below.

For active-LOW input

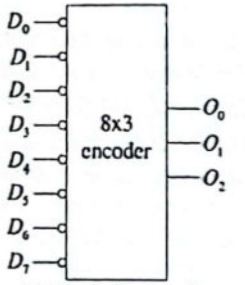
$$O_0 = D_1' + D_3' + D_5' + D_7' = (D_1 D_3 D_5 D_7)'$$

$$O_1 = D_2' + D_3' + D_6' + D_7' = (D_2 D_3 D_6 D_7)'$$

$$O_2 = D_4' + D_5' + D_6' + D_7' = (D_4 D_5 D_6 D_7)'$$

For active-HIGH input

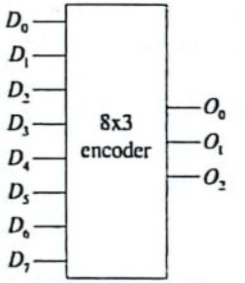
$$O_0 = D_1 + D_3 + D_5 + D_7$$
$$O_1 = D_2 + D_3 + D_6 + D_7$$
$$O_2 = D_4 + D_5 + D_6 + D_7$$



(a) Block diagram for active-LOW input

Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	O_2	O_1	O_0
1	1	1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	0	1	0	0	1
1	1	1	1	1	0	1	1	0	1	0
1	1	1	1	0	1	1	1	0	1	1
1	1	1	0	1	1	1	1	1	0	0
1	1	0	1	1	1	1	1	1	0	1
1	0	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1

(b) Truth table for active-LOW input



(c) Block diagram for active-HIGH input

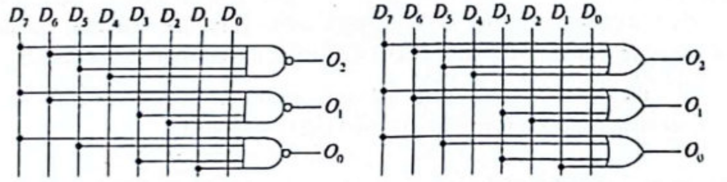
Inputs								Outputs		
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	O_2	O_1	O_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

(d) Truth table for active-HIGH input

Figure 8.1 Block diagram and truth table of octal-to-binary encoder (8-to-3-line encoder).

The logic implementation of the octal-to-binary encoder for both active-LOW and active-HIGH inputs is shown in Figure 8.2. Note that D_0 input is not connected to gates.

The encoder of Figure 8.2 assumes that only one input line can be activated at any time; otherwise the circuit has no meaning. Note that the circuit has eight inputs and could have $2^8 = 256$ possible input combinations. Only eight of these combinations have any meaning. The other input combinations are don't care conditions. Encoders of this type are not available in IC package, since they can be easily constructed with gates. The type of encoders available in IC form is called a *priority encoder*. These encoders establish an input priority to ensure that only the highest-priority input line is encoded. Thus, in truth tables of Figure 8.1, if priority is given to an input with higher subscript number over one with a lower subscript number, then if both D_2 and D_5 are activated simultaneously, the output will be 101 because D_5 has a higher priority over D_2. Of course, the truth table of a priority encoder is different from the one of Figure 8.1.


D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
O_2
O_1
O_0
D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0
O_2
O_1
O_0

(a) Active-LOW input (b) Active-HIGH input

Figure 8.2 Logic implementation of octal-to-binary encoder (8-to-3-line encoder).

8.2.2 Available MSI Encoder ICs

Available MSI priority encoder ICs are given in Table 8.1.

Table 8.1 Available MSI encoder ICs.

IC No.	Description
74147	Decimal-to-BCD priority encoder with active-LOW input and inverted outputs.
74148	Octal-to-binary priority encoder with active-LOW input and inverted outputs.

IC 74147 is introduced here. The pin assignment of IC 74147 is shown in Figure 8.3. IC 74147 9-input priority encoder accepts data from nine active-LOW inputs (D_1' - D_9') and provides a binary representation on the four active-LOW outputs (O_0' - O_3'). A priority is assigned to each input so that when two or more inputs are simultaneously active, the input with the highest priority is represented on the output, with input line D_9' having the highest priority. The IC provides the 10-to-4-line priority encoding function by use of the implied decimal "zero". The "zero" is encoded when all nine data inputs are HIGH, forcing all four outputs HIGH.

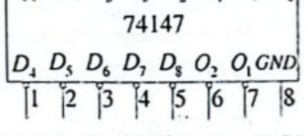

16 15 14 13 12 11 10 9
V_{CC} NC O_3 D_3 D_2 D_1 D_9 O_0
74147
D_4 D_5 D_6 D_7 D_8 O_2 O_1 GND
1 2 3 4 5 6 7 8

Figure 8.3 IC type 74147 Decimal-to-BCD priority encoder.

8.3 Decoders

8.3.1 Design of Decoders

A *decoder* is a combinational circuit that accepts n inputs that represent a binary code and activate only the output that corresponds to the input code. The number of output lines may be a maximum of 2^n. If the n-bit decoded information has unused or don't care combinations, the decoder output will have less than 2^n outputs. The output lines can be activated in two ways – by setting the activated output to 0 and the other outputs to 1 (active-LOW output) or by setting the activated output to 1 and the other outputs to 0 (active-HIGH output). If the number of output lines is less than 2^n, the decoder is often designed such that all the outputs remain inactivated when an invalid input combination (unused code) is applied.

Binary-to-octal decoder

The block diagrams and truth tables of a *binary-to-octal decoder* (also called *3-to-8-line decoder* or *1-of-8 decoder*) for both active-LOW and active-HIGH outputs are shown in Figure 8.4. From the truth tables, the Boolean functions for the outputs can be written as below.

For active-LOW output

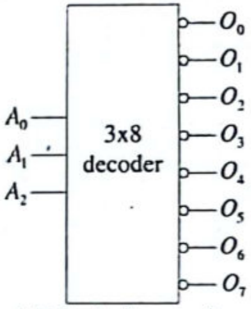
$$O_0 = A_2 + A_1 + A_0 = (A_2'A_1'A_0')'$$
$$O_1 = A_2 + A_1 + A_0' = (A_2'A_1'A_0)'$$
$$O_2 = A_2 + A_1' + A_0 = (A_2'A_1A_0')'$$
$$O_3 = A_2 + A_1' + A_0' = (A_2'A_1A_0)'$$
$$O_4 = A_2' + A_1 + A_0 = (A_2A_1'A_0')'$$
$$O_5 = A_2' + A_1 + A_0' = (A_2A_1'A_0)'$$
$$O_6 = A_2' + A_1' + A_0 = (A_2A_1A_0')'$$
$$O_7 = A_2' + A_1' + A_0' = (A_2A_1A_0)'$$

For active-HIGH output

$$O_0 = A_2'A_1'A_0'$$
$$O_1 = A_2'A_1'A_0$$
$$O_2 = A_2'A_1A_0'$$
$$O_3 = A_2'A_1A_0$$
$$O_4 = A_2A_1'A_0'$$

$$O_5 = A_2A_1'A_0$$
$$O_6 = A_2A_1A_0'$$
$$O_7 = A_2A_1A_0$$

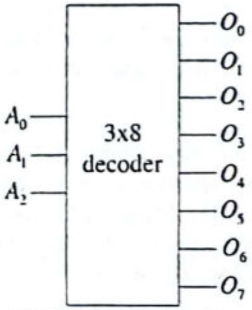
The logic implementation of the binary-to-octal decoder for both active-LOW and active-HIGH outputs is shown in Figure 8.5. Note that the outputs of the decoder with active-HIGH output are simply inverted to obtain the decoder with active-LOW output.



(a) Block diagram for active-LOW output

Inputs			Outputs							
A_2	A_1	A_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	0	1	1
0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	1	1	0	1	1	1	1
1	0	1	1	1	0	1	1	1	1	1
1	1	0	1	0	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1

(b) Truth table for active-LOW output

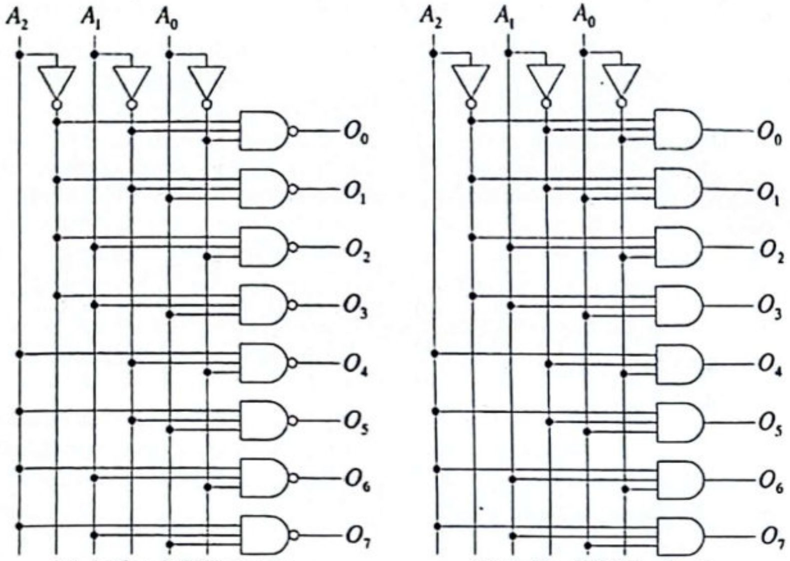


(c) Block diagram for active-HIGH output

Inputs			Outputs							
A_2	A_1	A_0	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

(d) Truth table for active-HIGH output

Figure 8.4 Block diagram and truth table of binary-to-octal decoder (3-to-8-line decoder or 1-of-8 decoder).



(a) Active-LOW output (a) Active-HIGH output

Figure 8.5 Logic implementation of binary-to-octal decoder (3-to-8-line decoder or 1-of-8 decoder).

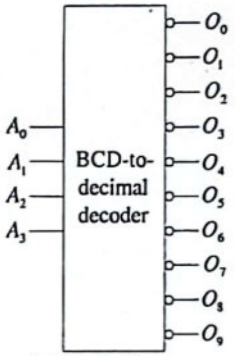
The decoders with binary inputs are called n-to-m line decoder (or 1-of-m decoders) where $m \leq 2^n$. Their purpose is to generate the 2^n (or less) minterms (active-HIGH output) or maxterms (active-LOW output) of n input variables.

BCD-to-decimal decoder

Sometimes decoders are designed to accept coded binary number such as BCD number or Excess-3 number. The design of a BCD-to-decimal decoder with active-LOW output is given here. The block diagram and truth table of a BCD-to-decimal decoder with active-LOW output are given in Figure 8.6. Note that for the invalid BCD inputs (1010 to 1111) no output line is activated. The Boolean functions for the outputs can be written from the truth table as

$$O_0 = A_3 + A_2 + A_1 + A_0 = (A_3' A_2' A_1' A_0')'$$
$$O_1 = A_3 + A_2 + A_1 + A_0' = (A_3' A_2' A_1' A_0)'$$
$$O_2 = A_3 + A_2 + A_1' + A_0 = (A_3' A_2' A_1 A_0')'$$
$$O_3 = A_3 + A_2 + A_1' + A_0' = (A_3' A_2' A_1 A_0)'$$
$$O_4 = A_3 + A_2' + A_1 + A_0 = (A_3' A_2 A_1' A_0')'$$
$$O_5 = A_3 + A_2' + A_1 + A_0' = (A_3' A_2 A_1' A_0)'$$

$$O_6 = A_3 + A_2' + A_1' + A_0 = (A_3' A_2 A_1 A_0')'$$
$$O_7 = A_3 + A_2' + A_1' + A_0' = (A_3' A_2 A_1 A_0)'$$
$$O_8 = A_3' + A_2 + A_1 + A_0 = (A_3 A_2' A_1' A_0')'$$
$$O_9 = A_3' + A_2 + A_1 + A_0' = (A_3 A_2' A_1' A_0)'$$



(a) Block diagram

Inputs				Outputs									
A_3	A_2	A_1	A_0	O_9	O_8	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	1	1	1	1	1	1	1	1	1	0
0	0	0	1	1	1	1	1	1	1	1	1	0	1
0	0	1	0	1	1	1	1	1	1	1	0	1	1
0	0	1	1	1	1	1	1	1	1	0	1	1	1
0	1	0	0	1	1	1	1	1	0	1	1	1	1
0	1	0	1	1	1	1	1	0	1	1	1	1	1
0	1	1	0	1	1	1	0	1	1	1	1	1	1
0	1	1	1	1	1	0	1	1	1	1	1	1	1
1	0	0	0	1	0	1	1	1	1	1	1	1	1
1	0	0	1	0	1	1	1	1	1	1	1	1	1
1	0	1	0	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1

(b) Truth table

Figure 8.6 Block diagram and truth table of BCD-to-decimal decoder with active-LOW output.

The logic implementation of the BCD-to-decimal decoder with active-LOW output is shown in Figure 8.7.

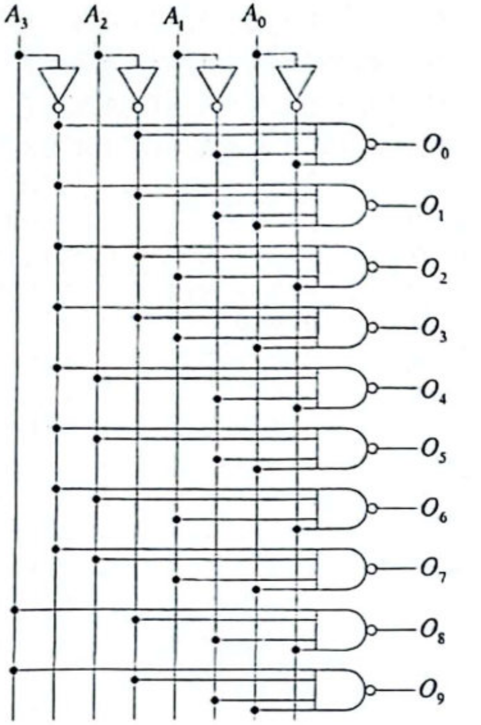


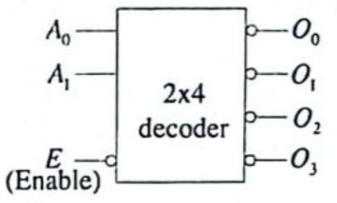
Figure 8.7 Logic implementation of a BCD-to-decimal decoder with active-LOW output.

8.3.2 Decoders with Enable Input

Some decoders have enable input that is used to control the operation of the decoder. If the enable input is active, the output corresponding to the input is activated. If the enable input is not active, no output is activated. The block diagram and truth table of a 2-to-4-line decoder with active-LOW output and active-LOW enable is shown in Figure 8.8. From truth table the Boolean functions for the outputs can be written as

$$O_0 = E + A_1 + A_0 = (E'A_1'A_0')'$$
$$O_1 = E + A_1 + A_0' = (E'A_1'A_0)'$$
$$O_2 = E + A_1' + A_0 = (E'A_1A_0')'$$
$$O_3 = E + A_1' + A_0' = (E'A_1A_0)'$$

The logic implementation of the 2-to-4-line decoder with active-LOW output and active-LOW enable input is shown in Figure 8.9.



(a) Block diagram

Inputs			Outputs			
E	A_1	A_0	O_3	O_2	O_1	O_0
0	0	0	1	1	1	0
0	0	1	1	1	0	1
0	1	0	1	0	1	1
0	1	1	0	1	1	1
1	0	0	1	1	1	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

(b) Truth table

Figure 8.8 Block diagram and truth table of 2-to-4-line decoder with active-LOW output and active-LOW enable input.

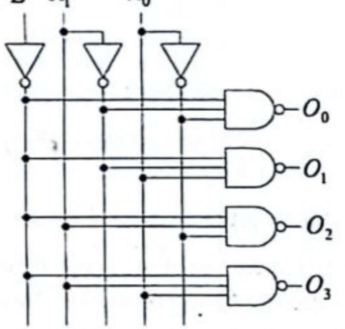
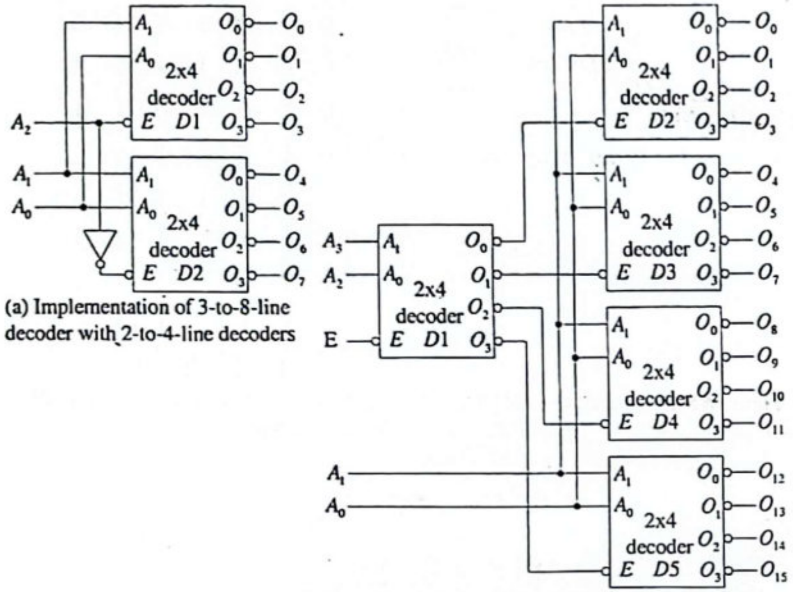


Figure 8.9 Logic implementation of 2-to-4-line decoder with active-LOW output and active-LOW enable input.

8.3.3 Expansion of Decoder

To meet the larger inputs needs, decoders can be expanded by using *enable* input line. Figure 8.10(a) shows a 3-to-8-line decoder and Figure 8.10(b) shows a 4-to-16-line decoder using 2-to-4-line decoders.

In Figure 8.10(a), when $A_2 = 0$, decoder $D1$ is enabled and decoder $D2$ is disabled. The output corresponding to A_1A_0 input of the decoder $D1$ is activated. When $A_2 = 1$, decoder $D2$ is enabled and decoder $D1$ is disabled. The output corresponding to A_1A_0 input of the decoder $D2$ is activated. For example, if $A_2A_1A_0 = 101$, decoder $D2$ will be enabled since $A_2 = 1$ and the output O_1 of the decoder *D*2 will be activated since $A_1A_0 = 01$. Therefore, output O_5 of the combined decoder will be activated.



(a) Implementation of 3-to-8-line decoder with 2-to-4-line decoders

(b) Implementation of 4-to-16-line decoder with 2-to-4-line decoders

Figure 8.10 Expansion of decoder.

In Figure 8.10(b), when high-order bits $A_3A_2 = 00$, the output O_0 of the decoder *D*1 will be LOW and the decoder *D*2 will be enabled. The output of the decoder *D*2 corresponding to the low-order bits A_1A_0 will be activated. Similarly, when $A_3A_2 = 01$, the decoder *D*3 will be enabled and the output of the decoder *D*3 corresponding to A_1A_0 will be activated. When $A_3A_2 = 10$, decoder *D*4 will be enabled and the output of the decoder *D*4 corresponding to A_1A_0 will be activated. Finally, when $A_3A_2 = 11$, decoder *D*5 will be enabled and the output of the decoder *D*5 corresponding to A_1A_0 will be activated. For example, when $A_3A_2A_1A_0 = 0110$, that is high-order bits $A_3A_2 = 01$ and low-order bits $A_1A_0 = 10$, the O_1 output of the decoder *D*1 will be LOW, which will enable the decoder *D*3. Then the output O_2 of the decoder *D*3 will be activated, which is the output O_6 of the combined decoder.

8.3.4 Combinational Logic Implementation with Decoder

A decoder provides the 2^n minterms (active-HIGH output) or maxterms (active-LOW output) of *n* input variables. Since, any Boolean function can be expressed as sum of minterms (canonical sum of products) or product of maxterms (canonical product of sums), decoders can be used to implement Boolean function in conjunction with some external gates. In this implementation method, any combinational circuit with *n* inputs and *m* outputs can be implemented with an *n*- to-2^n-line decoder and *m* OR/NAND or AND/NOR gates.

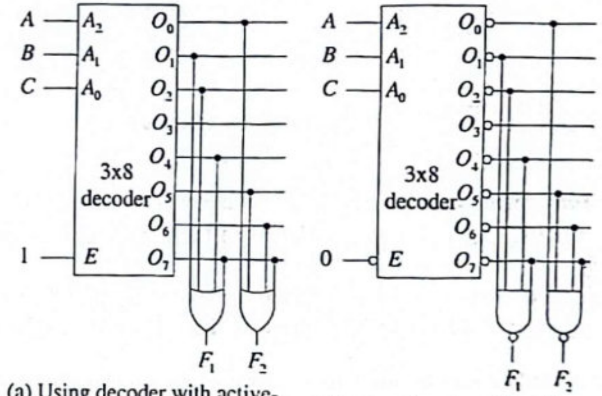
Implementation of Canonical Sum of Products Function

For implementing an *n*-variable canonical sum of products function, the *n* inputs of the function are applied to the *n* inputs of an *n*- to-2^n-line decoder with active-HIGH outputs. As the outputs of the decoder generate the minterms of an *n*-variable function, the outputs of the decoder corresponding to the minterms of the function are ORed with an OR gate to form the sum. Implementation of the following 3-variable functions using a 3-to-8-line decoder with active-HIGH output is shown in Figure 8.11(a).

$$F_1(A,B,C) = \sum(1,2,4,7)$$

$$F_2(A,B,C) = \sum(0,5,6,7)$$

The canonical sum of products function can also be implemented using decoder with active-LOW output. In this decoder, the outputs represent the complement of the minterms. In this implementation, the complemented minterms are combined with a NAND gate to form the sum of minterms. The procedure is obvious from DeMorgan's theorem. Implementation of the above functions using a 3-to-8-line decoder with active-LOW output is shown in Figure 8.11(b).



(a) Using decoder with active-HIGH outpt

(a) Using decoder with active-LOW outpt

Figure 8.11 Implementation of canonical sum of product functions $F_1(A,B,C) = \sum(1,2,4,7)$ and $F_2(A,B,C) = \sum(0,5,6,7)$ using decoders.

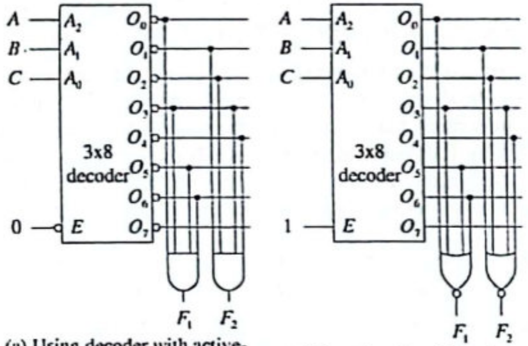
Implementation of Canonical Product of Sums Function

For implementing an n-variable canonical product of sums function, the n inputs of the function are applied to the n inputs of an n-to-2^n-line decoder with active-LOW outputs. As the outputs of the decoder generate the maxterms of an n-variable function, the outputs of the decoder corresponding to the maxterms of the function are ANDed with an AND gate to form the product. Implementation of the following 3-variable functions using a 3-to-8-line decoder with active-LOW output is shown in Figure 8.12(a).

$$F_1(A,B,C)=\prod(0,3,5,6)$$

$$F_2(A,B,C)=\prod(1,2,3,4)$$

The canonical product of sums function can also be implemented using decoders with active-HIGH outputs. In this decoder, the outputs represent the complement of the maxterms. In this implementation, the complemented maxterms are combined with a NOR gate to form the product of maxterms. The procedure is obvious from DeMorgan's theorem. Implementation of the above functions using a 3-to-8-line decoder with active-HIGH output is shown in Figure 8.12(b).



(a) Using decoder with active-LOW outpt

(a) Using decoder with active-HIGH outpt

Figure 10.12 Implementation of canonical product of sums function $F_1(A,B,C)=\prod(0,3,5,6)$ and $F_2(A,B,C)=\prod(1,2,3,4)$ using decoders.

If the number of minterms in a function is less than $2^n/2$, then the implementation of Figure 8.11 is suitable, otherwise the implementation of Figure 8.12 is suitable.

The decoder method can be used to implement any combinational circuit. However, its implementation must be compared with all other possible implementations to determine the best solution. In some cases this method may provide the best implementation, especially if the combinational circuit has many outputs and if each output function is expressed with a small number of minterms or maxterms.

8.3.5 Available MSI Decoder ICs

List of available MSI decoder ICs are given in Table 8.2.

Table 8.2 Available MSI decoder ICs.

IC No.	Description
74139	Dual 2-to-4-line decoder with active-LOW output
74155	Dual 2-to-4-line decoder with active-LOW output
74156	Dual 2-to-4-line decoder with active-LOW output (open-collector)
74138	3-to-8-line decoder with active-LOW output
74154	4-to-16-line decoder with active-HIGH output
7442	BCD-to-decimal decoder with active-LOW output.

IC 74138 is introduced here. The pin assignment of IC 74138 is shown in Figure 8.13. A_0 through A_2 are three address lines and O_0' through O_7' are eight active-LOW outputs. It has two active-LOW (E_1' and E_2') and one active-HIGH (E_3) enable inputs internally ANDed to produce a single enable signal.

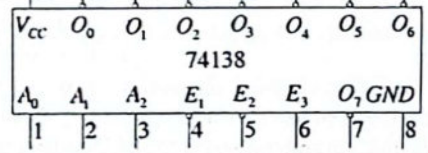


Figure 8.13 IC type 74138 3-to-8-line decoder with active-LOW output.

8.4 Code Converters

There are a wide variety of binary codes used in digital systems. Some of these codes are BCD, Excess-3, etc. Different digital systems use different codes. It is sometimes necessary to use the output of one system as the input to another. Therefore, it is required to convert from one code to another. This sort of code converters are combinational circuits and may be regarded as special type of encoders, which encodes from one form to another. Code converters can be designed using classical method of combinational logic design or using MSI circuits. Design of BCD-to-Excess-3 code converter is given here.

The block diagram and truth table for BCD-to-Excess-3 code converter is shown in Figure 8.14. Note that the input combinations 1010 to 1111 are invalid BCD code and will not appear in the input. Therefore, their outputs are don't care states. The output functions are simplified using K-maps in Figure 8.15, which gives

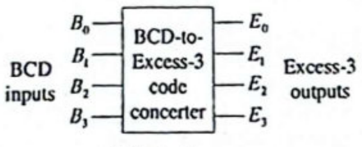
$$E_0=B_0'$$

$$E_1=B_1'B_0'+B_1B_0$$

$$E_2=B_2'B_0+B_2'B_1+B_2B_1'B_0'$$

$$E_3=B_3+B_2B_0+B_2B_1$$

The logic implementation for BCD-to-Excess-3 code converter is shown in Figure 8.16.

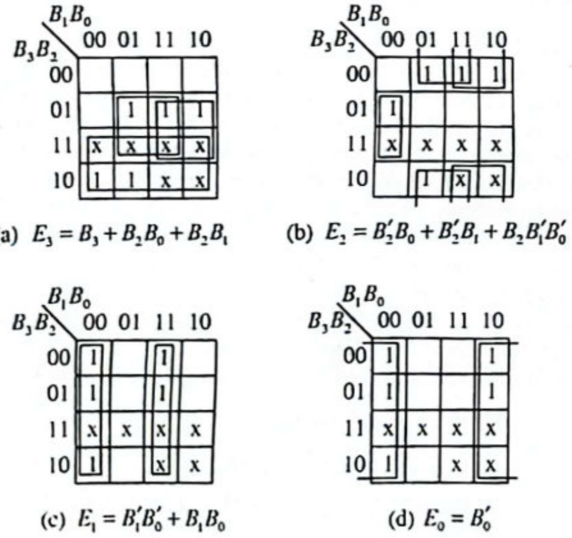

BCD inputs
B_0
B_1
B_2
B_3
BCD-to-Excess-3 code concerter
E_0
E_1
E_2
E_3
Excess-3 outputs

(a) Block diagram

Input BCD				Output Excess-3			
B_3	B_2	B_1	B_0	E_3	E_2	E_1	E_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	x	x	x	x
1	0	1	1	x	x	x	x
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

(b) Truth table

Figure 8.14 Block diagram and truth table for BCD-to-Excess-3 code converter.


B_1B_0
B_3B_2
00 01 11 10
00
01
11
10

(a) $E_3 = B_3 + B_2B_0 + B_2B_1$

(b) $E_2 = B_2'B_0 + B_2'B_1 + B_2B_1'B_0'$

(c) $E_1 = B_1'B_0' + B_1B_0$

(d) $E_0 = B_0'$

Figure 8.15 K-maps for BCD-to-Excess-3 code converter.

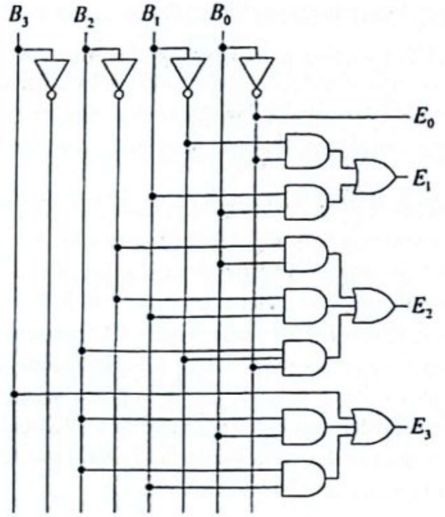

B_3
B_2
B_1
B_0
E_0
E_1
E_2
E_3

Figure 8.16 Logic implementation for BCD-to-Excess-3 code converter.

The BCD-to-Excess-3 code converter can be implemented using a 4-bit binary adder. From the truth table of Figure 8.14(b), it can be noted that Excess-3 code can be obtained by adding binary 3 (0011) with the BCD code. This addition can be implemented by a 4-bit binary adder as shown in Figure 8.17.

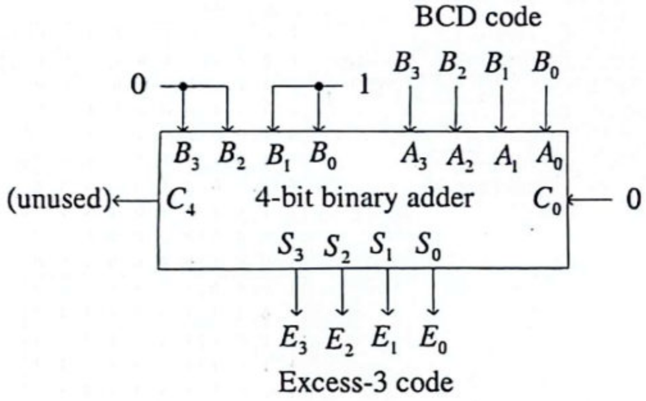

BCD code
B_3 B_2 B_1 B_0
0
1
B_3 B_2 B_1 B_0
A_3 A_2 A_1 A_0
(unused)
C_4
4-bit binary adder
C_0
0
S_3 S_2 S_1 S_0
E_3 E_2 E_1 E_0
Excess-3 code

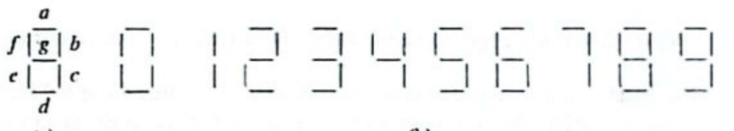
Figure 8.17 BCD-to-Excess-3 code converter using 4-bit binary adder.

8.5 BCD-to-7-Segment Display Decoder

8.5.1 Design of BCD-to-7-Segment Display Decoder

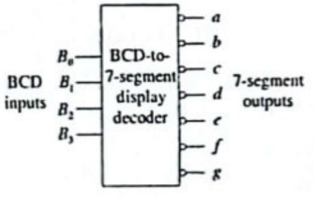
In many digital systems, we prefer to see the output in a decimal format. Seven segment display is the most popular display device used in digital systems. For displaying data using this device, the data have to be converted from BCD to 7-segment code.

In 7-segment display seven light-emitting diodes (LEDs) are arranged as in Figure 8.18(a). By controlling the current through each LED, some segments will be light and others will be dark so that the desired digit pattern will be generated. Figure 8.18(b) shows the segment patterns for digits 0 to 9. For example, to display a "6", the segments *a, c, d, e, f, g* are made bright while segment *b* is dark. A BCD-to-7-segment display decoder is used to take a 4-bit BCD input and to activate the outputs that will pass current through the appropriate segments to display the decimal digit. The block diagram and truth table of a BCD-to-7-segment decoder with active-LOW outputs is shown in Figure 8.19. Note that for the invalid BCD inputs (1010 to 1111) no segment will be activated.



(a) (b)

Figure 8.18 (a) 7-segment arrangement and (b) activated segments for digits 0 to 9.



(a) Block diagram

BCD inputs				7-segment outputs						
B_3	B_2	B_1	B_0	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
0	0	1	1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0
1	0	1	0	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1

(b) Truth table

Figure 8.19 Block diagram and truth table of BCD-to-7-segment display decoder with active-LOW output.

The output functions are simplified using K-maps in Figure 8.20, which gives

$$a = B_3B_2 + B_3B_1 + B_2B_1'B_0' + B_3'B_2'B_1'B_0$$

$$b = B_3B_2 + B_3B_1 + B_2B_1'B_0 + B_2B_1B_0'$$

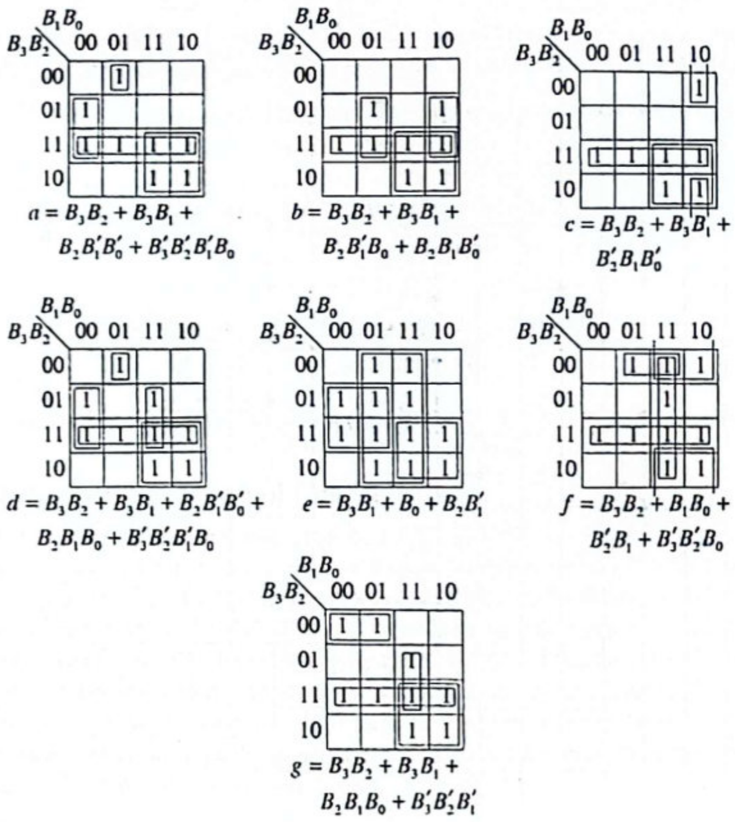
$$c = B_3B_2 + B_3B_1 + B_2'B_1B_0'$$

$$d = B_3B_2 + B_3B_1 + B_2B_1'B_0' + B_2B_1B_0 + B_3'B_2'B_1'B_0$$

$$e = B_3B_1 + B_0 + B_2B_1'$$

$$f = B_3B_2 + B_1B_0 + B_2'B_1 + B_3'B_2'B_0$$

$$g = B_3B_2 + B_3B_1 + B_2B_1B_0 + B_3'B_2'B_1'$$



$a = B_3B_2 + B_3B_1 + B_2B_1'B_0' + B_3'B_2'B_1'B_0$

$b = B_3B_2 + B_3B_1 + B_2B_1'B_0 + B_2B_1B_0'$

$c = B_3B_2 + B_3B_1 + B_2'B_1B_0'$

$d = B_3B_2 + B_3B_1 + B_2B_1'B_0' + B_2B_1B_0 + B_3'B_2'B_1'B_0$

$e = B_3B_1 + B_0 + B_2B_1'$

$f = B_3B_2 + B_1B_0 + B_2'B_1 + B_3'B_2'B_0$

$g = B_3B_2 + B_3B_1 + B_2B_1B_0 + B_3'B_2'B_1'$

Figure 8.20 K-maps for BCD-to-7-segment display decoder with active-LOW output.

The logic implementation of the BCD-to-7-segment display decoder with active-LOW output is shown in Figure 8.21. Note that B_3B_2 is common to all outputs but *e*; B_3B_1 is common to all outputs but *f*; $B_3'B_2'B_1'B_0$ is common to *a* and *d* outputs; and $B_2B_1B_0$ is common to *d* and *g* outputs; $B_2B_1'B_0$ is common to *a* and *d* outputs.

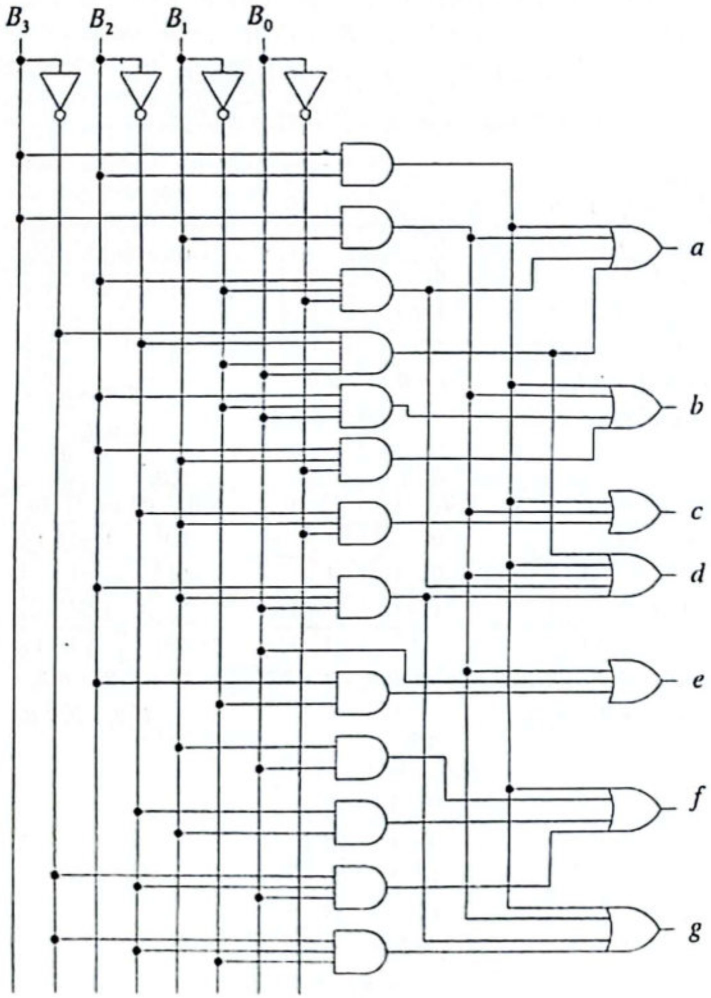


Figure 8.21 Logic implementation for BCD-to-7-segment display decoder with active-LOW output.

8.5.2 Available MSI BCD-to-7-Segment Display Decoder/Driver ICs

Available MSI BCD-to-7-segment display decoder/driver ICs are given in Table 8.3. Note that these MSI ICs also have LED driving capabilities.

Table 8.3 Available MSI BCD-to-7-segment display decoder/driver ICs.

IC No	Output
7446, 74246	Active-LOW; Open collector
7447, 74247	Active-LOW; Open collector
7448, 74248	Active-HIGH; Pull-up resistor = 2kΩ
7449, 74249	Active-HIGH; Open collector

IC 7447 is introduced here. The pin assignment of IC 7447 is shown in Figure 8.22. The IC 7447 accepts four lines of BCD input data ($B_0 - B_3$) and generates the corresponding seven-segment code with active-LOW, open collector outputs ($a' - g'$). If the input data is decimal zero, $RBI' = 0$ blanks the display. $BI / RBO' = 0$ unconditionally turns off all segment outputs. $LT' = 0$ turns on all segment outputs, provided that $BI / RBO' = 1$.

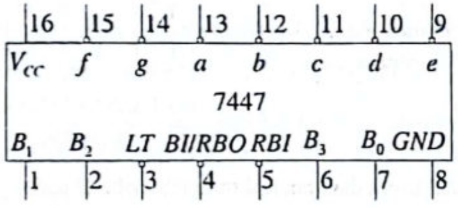


Figure 8.22 IC type 7447 BCD-to-7-segment display decoder/driver with active-LOW, open collector output.

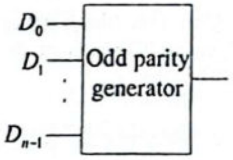
8.6 Parity Generator and Checker

8.6.1 Design of Parity Generator and Checker

As discussed in Chapter 2, a parity bit is an encoding scheme for detecting errors during transmission of binary information. A parity bit is an extra bit included with a binary message to make the number of 1s either odd (odd parity) or even (even parity). The message, including the parity bit, is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond to the one transmitted. The circuit that generates the parity bit in the transmitter is called a *parity generator*; the circuit that checks the parity in the receiver is called a *parity checker*.

Odd Parity Generator

Block diagram of an *n*-bit odd parity generator and its truth table for 3-bit and 4-bit messages are shown in Figure 8.23.



(a) Block diagram

D_2	D_1	D_0	P
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(b) Truth table for 3-bit message

D_3	D_2	D_1	D_0	P
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(c) Truth table for 4-bit message

Figure 8.23 Block diagram and truth table of odd parity generator.

From the truth table of Figure 8.23(b), we see that the function for the output P consists of four minterms with numerical values having an even number of 1s. This is the complement of EXOR function of the inputs. Remember that for an odd number of inputs EXOR function and EXNOR function are equal. So, it can also be said that the function for the output P is the complement of EXNOR function of the inputs. From this observation, the function for the output P can be written as

$$P = (D_2 \oplus D_1 \oplus D_0)'$$

From the truth table of Figure 8.23(c), we see that the function for the output P consists of eight minterms with numerical values having an even number of 0s. This is the EXNOR function of the inputs. Remember that for an even number of inputs EXOR function and EXNOR function are complement of each other. So, it can also be said that the function for the output P is the complement of EXOR function of the inputs. From this observation, the function for the output P can be written as

$$P = (D_3 \oplus D_2 \oplus D_1 \oplus D_0)'$$

From the above observations, we see that the function for the output P for an n number of inputs is the complement of the EXOR function of the inputs, that is,

$$P = (D_{n-1} \oplus D_{n-2} \oplus \cdots \oplus D_1 \oplus D_0)'$$

Logic implementation of an odd parity generator for 4-bit message is shown in Figure 8.24.

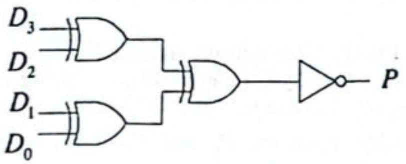


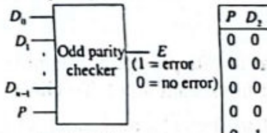
Figure 8.24 Logic implementation for odd parity generator for 4-bit message.

Odd Parity Checker

Block diagram of an odd parity checker for n-bit message ($n-1$ bit data plus a parity bit) and its truth table for 3-bit message and 4-bit message are shown in Figure 8.25. The truth tables of Figure 8.25(b) and (c) are similar to the truth tables of Figure 8.23(b) and (c). Following the same reasoning, the function for the output E of an odd parity checker for an n-bit message can be written as

$$E = (P \oplus D_{n-1} \oplus \cdots \oplus D_1 \oplus D_0)'$$

Logic implementation of an odd parity checker for 4-bit message is shown in Figure 8.26. From Figure 8.24 and Figure 8.26, we see that the same circuit can be used as both parity generator and parity checker.



(a) Block diagram

P	D_1	D_0	E
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

(b) Truth table for 3-bit message

P	D_2	D_1	D_0	E
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(c) Truth table for 4-bit message

Figure 8.25 Block diagram and truth table of odd parity checker.

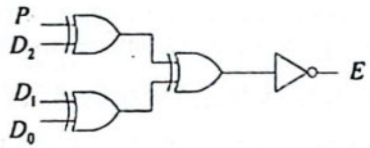


Figure 8.26 Logic implementation for odd parity checker for 4-bit message.

8.6.2 Available MSI Parity Generator/Checker IC

MSI IC type 74180 is an 8-bit parity generator/checker, which generates and checks both the odd and even parity. Pin assignment of IC 74180 is shown in Figure 8.27. Even and odd parity enable inputs are P_E and P_O respectively. Data inputs are $D_0 - D_7$ and even and odd parity outputs are E_E and E_O respectively.

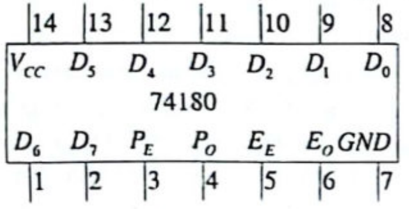

14 13 12 11 10 9 8
V_{CC} D_5 D_4 D_3 D_2 D_1 D_0
74180
D_6 D_7 P_E P_O E_E E_O GND
1 2 3 4 5 6 7

Figure 8.27 IC type 74180 8-bit parity generator/checker.

True active-HIGH or true active-LOW parity can be generated at both E_E and E_O outputs. True active-HIGH parity is established with enable inputs $P_E = 1$ and $P_O = 0$. True active-LOW parity is established when enable inputs $P_E = 0$ and $P_O = 1$.

Parity checking of a 9-bit word (8 bits plus parity) is possible by using the two enable inputs plus an inverter as the ninth data input. To check for true active-HIGH parity, the ninth data input is tied to the P_O input and an inverter is connected between the P_O and P_E inputs. To check for true active-LOW parity, the ninth data input is tied to the P_E input and an inverter is connected between the P_E and P_O inputs.

8.7 Multiplexers

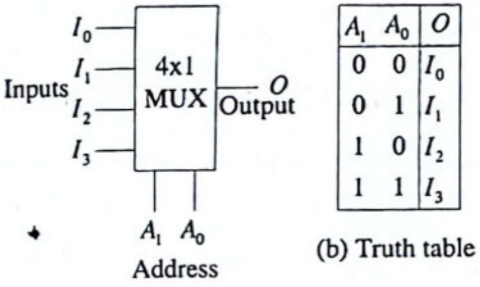
8.7.1 Design of Multiplexers

A digital multiplexer (MUX in short) is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of address lines (also called selection lines). Normally, there are 2^n input lines and n address lines whose bit combinations determine which input is selected.

The block diagram and truth table of a 4-to-1-line multiplexer is shown in Figure 8.28. From the truth table of Figure 8.28(b), the Boolean function for the output O can be written as

$$O = I_0 A_1' A_0' + I_1 A_1' A_0 + I_2 A_1 A_0' + I_3 A_1 A_0$$

The logic implementation for the 4-to-1-line multiplexer is shown in Figure8.29.



A_1	A_0	O
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(a) Block diagram (b) Truth table

Figure 8.28 Block diagram and truth table of a 4-to-1-line multiplexer.

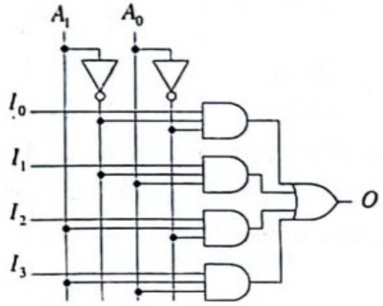


Figure 8.29 Logic implementation for a 4-to-1-line multiplexer.

A multiplexer is also called a data selector, since it selects one of many inputs and steers the binary information to the output line.

The AND gate portion of the multiplexer resembles a decoder circuit and, indeed, they decode the input address lines. In general, a 2^n-to-1-line multiplexer is constructed from an n-to-2^n-line decoder by adding to it 2^n input lines, one to each AND gate. The outputs of the AND gates are applied to a single OR gate to provide the 1-line output. The size of a multiplexer is specified by the number of 2^n of its input lines and the single output line. It is then implied that it also contains n address lines.

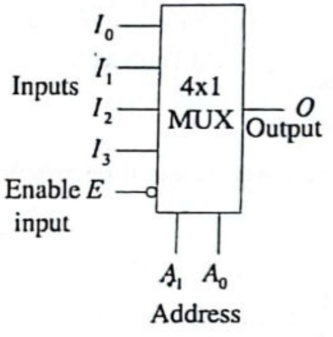
8.7.2 Multiplexer with Enable Input

Multiplexers may have an enable input to control the operation of the circuit. When the enable input is in active state, the circuit functions as a normal multiplexer, and when it is in inactive state, the outputs are disabled. Normally, the enable input is active-LOW.

The block diagram and truth table of a 4-to-1-line multiplexer with active-LOW enable input is given in Figure 8.30.From the truth table of Figure 8.30(b), the Boolean function for the output O can be written as

$$O = I_0E'A_1'A_0' + I_1E'A_1'A_0 + I_2E'A_1A_0' + I_3E'A_1A_0$$

The logic implementation for the 4-to-1-line multiplexer is shown in Figure 8.31.



(a) Block diagram

E	A_1	A_0	O
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

(b) Truth table

Figure 8.30 Block diagram and truth table of a 4-to-1-line multiplexer with active-LOW enable input.

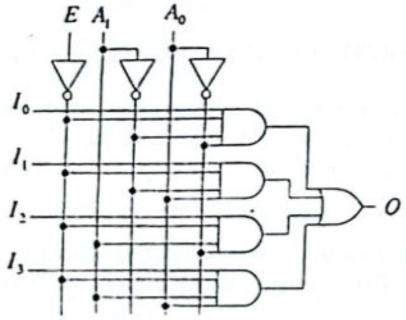


Figure 8.31 Logic implementation of a 4-to-1-line multiplexer with active-LOW enable input.

8.7.3 Available MSI Multiplexer ICs

Available MSI multiplexer ICs are given in Table 8.4.

Table 8.4 Available MSI multiplexer ICs.

IC No	Description	Output
74157	Quad 2×1 multiplexer	Same as input
74158	Quad 2×1 multiplexer	Inverted input
74153	Dual 4×1 multiplexer	Same as input
74352	Dual 4×1 multiplexer	Inverted input
74151	8×1 multiplexer	Complementary outputs
74150	16×1 multiplexer	Inverted input

IC type 74151 is illustrated here. IC 74151 is an 8-to-1-line MUX with active-LOW enable input. The pin assignment of the IC is shown in Figure 8.32. The eight inputs are designated I_0 through I_7. The three address or select lines $A_2A_1A_0$ select the particular input to be multiplexed and applied to the output. The IC provides both uncomplemented (O) and complemented (O') output. The IC also has an active-LOW enable input E.

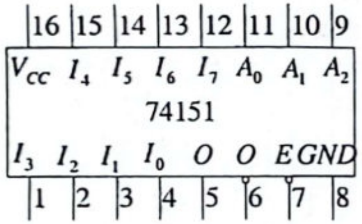


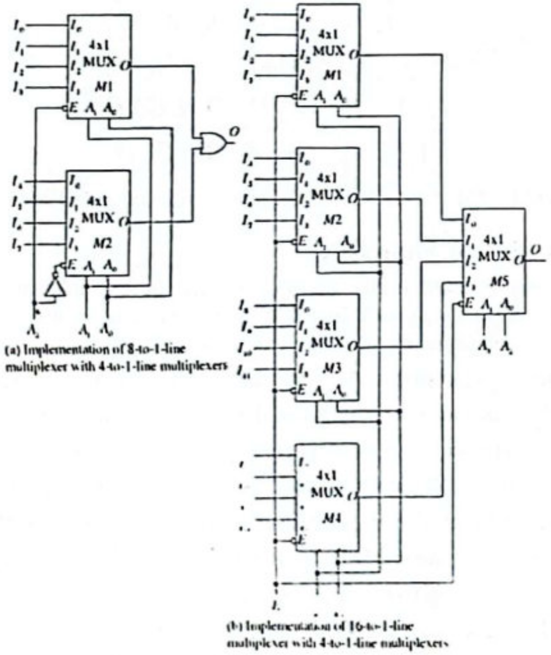
Figure 8.32 IC 74151 8-to-1-line multiplexer with active-LOW enable input.

8.7.4 Expansion of Muliplexer

To meet the larger inputs needs, multiplexers can be expanded by using enable input. Figure 8.33(a) shows an 8-to-1-line multiplexer and Figure 8.33(b) shows a 16-to-1-line multiplexer using 4-to-1-line multiplexers.

In Figure 8.33(a), when $A_2 = 0$, the multiplexer $M1$ is enabled and the multiplexer $M2$ is disabled. The output of the multiplexer $M2$ is 0 and the output of the multiplexer $M1$ is equal to its input corresponding to its A_1A_0 address bits. So, the final output of the OR gate is equal to the selected input of the multiplexer $M1$. When $A_2 = 1$, the multiplexer $M1$ is disabled and the multiplexer $M2$ is enabled. The output of the multiplexer $M1$ is 0 and the output of the multiplexer $M2$ is equal to its input corresponding to its A_1A_0 address bits. So, the final output of the OR gate is equal to the selected input of the multiplexer $M2$. For example, if $A_2A_1A_0 = 101$, that is $A_2 = 1$ and $A_1A_0 = 01$, then the multiplexer $M2$ will be enabled and its I_1 input, which is I_5 input of the combined multiplexer, is selected and the final output will be equal to this input.

In Figure 8.33(b), when higher-order address bits $A_3A_2 = 00$, the output of the multiplexer *M5* is equal to the output of the multiplexer *M1*. The output of the multiplexer *M1* is equal to its input corresponding to the low-order address bits A_1A_0. Thus, the output of the multiplexer *M5* is equal to the input of the multiplexer *M1* corresponding to A_1A_0. Similarly, when $A_3A_2 = 01$, the output of the multiplexer *M5* is equal to the input of the multiplexer *M2* corresponding to A_1A_0. When $A_3A_2 = 10$, the output of the multiplexer *M5* is equal to the input of the multiplexer *M3* corresponding to A_1A_0. Finally, when $A_3A_2 = 11$, the output of the multiplexer *M5* is equal to the input of the multiplexer *M4* corresponding to A_1A_0. For example, when $A_3A_2A_1A_0 = 0110$, that is $A_3A_2 = 01$ and $A_1A_0 = 10$, the output of the multiplexer *M5* will be equal to its I_1 input. The input I_1 of the multiplexer *M5* will be equal to the output of the multiplexer *M2*. The output of the multiplexer *M2* will be equal to its input I_2, which is the input I_6 of the combined multiplexer.



(a) Implementation of 8-to-1-line multiplexer with 4-to-1-line multiplexers

(b) Implementation of 16-to-1-line multiplexer with 4-to-1-line multiplexers

Figure 8.33 Expansion of multiplexers.

8.7.5 Combinational Logic Implementation with Multiplexers

It was shown in Section 8.3 that a decoder with active-HIGH output could be used to implement a Boolean function by employing an external OR gate. A quick reference to the multiplexer of Figure 8.29 reveals that it is essentially a decoder with active-HIGH output with OR gate already available. The minterms out of the decoder to be chosen can be controlled with the input lines. The minterm to be included with the function being implemented are chosen by making their corresponding input lines equal to 1, those minterms not included in the function are disabled by making their input lines equal to 0. The input variables of the function are applied to the address lines. This gives a method for implementing any Boolean function of *n* variables with a 2^n-to-1-line multiplexer. The implementation of the function

$$F(A,B,C,D) = \sum(0,2,3,6,8,9,12,14)$$

using a 16×1 multiplexer is shown in Figure 8.34.

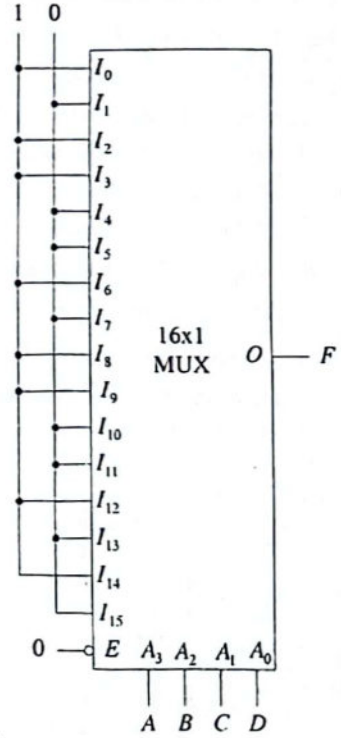


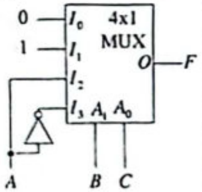
Figure 8.34 Implementation of the function $F(A,B,C,D) = \sum(0,2,3,6,8,9,12,14)$ using 16×1 multiplexer.

It is possible to do better than the above procedure. If we have a Boolean function of $n+1$ variables, we take n of these variables and connect them to the address lines of a multiplexer. The remaining single variable of the function is used for the inputs of the multiplexer. If A is this single variable, the inputs of the multiplexer are chosen to be either A or A' or 1 or 0. By judicious use of these four values for the inputs and by connecting the other variables to the address lines, we can implement any Boolean function with a multiplexer. In this way, it is possible to generate any function of $n+1$ variables with a 2^n-to-1-line multiplexer.

To demonstrate this procedure with a concrete example, consider the following function of three variables

$$F(A,B,C)=\sum(1,3,5,6)$$

The function can be implemented with a 4-to-1-line multiplexer as shown in Figure 8.35. Two of the variables, B and C are applied to the address lines in that order, i.e., B is connected to A_1 and C to A_0. The inputs of the multiplexer are 0, 1, A, and A'. When $BC=00$, output $F=0$ since $I_0=0$. Therefore, both minterms $m_0=A'B'C'$ and $m_4=AB'C'$ produce a 0 output, since the output is 0 when $BC=00$ regardless of the value of A. When $BC=01$, output $F=1$, since $I_1=1$. Therefore, both minterms $m_1=A'B'C$ and $m_5=AB'C$ produce a 1 output, since the output is 1 when $BC=01$ regardless of the value of A. When $BC=10$, input I_2 is selected. Since A is connected to this input, the output will be equal to 1 only for minterm $m_6=ABC'$, but not for minterm $m_2=A'BC'$, because when $A'=1$, then $A=0$, and since $I_2=0$, we have $F=0$. Finally, when $BC=11$, input I_3 is selected. Since A' is connected to this input, the output will be equal to 1 only for minterm $m_3=A'BC$, but not for $m_7=ABC$. This information is summarized in Figure 8.35(b), which is the truth table of the function we want to implement.



(a) Implementation with multiplexer

Minterm	A	B	C	F
0	0	0	0	0
1	0	0	1	1
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	0

(b) Truth table

	I_0	I_1	I_2	I_3
A'	0	①	2	③
A	4	⑤	⑥	7
	0	1	A	A'

(c) Implementation table

Figure 8.35 Implementation of the function $F(A,B,C)=\sum(1,3,5,6)$ with a multiplexer.

The above discussion shows by analysis that the multiplexer implements the required function. We now present a general procedure for implementing any Boolean function of n variables with a 2^{n-1}-to-1 multiplexer.

First, express the function in its sum of minterms form. Assume that the ordered sequence of variables chosen for the minterms is $ABCD\cdots$, where A is the leftmost variable in the ordered sequence of n variables and $BCD\cdots$ are the remaining $n-1$ variables. Connect the $n-1$ variables to the address lines of the multiplexer with B connected to the high-order address line, C to the next lower address line, and so on down to the last variable, which is connected to the lowest-order line A_0. Consider now the single variable A. Since this variable is in the highest-order position in the sequence of variables, it will be complemented in minterms 0 to $(2^n/2)-1$ which comprise the first half in the list of minterms. The second half of the minterms will have their A variable uncomplemented. For a three-variable function, A, B, C, we have eight minterms. Variable A is complemented in minterms 0 to 3 and uncomplemented in minterms 4 to 7.

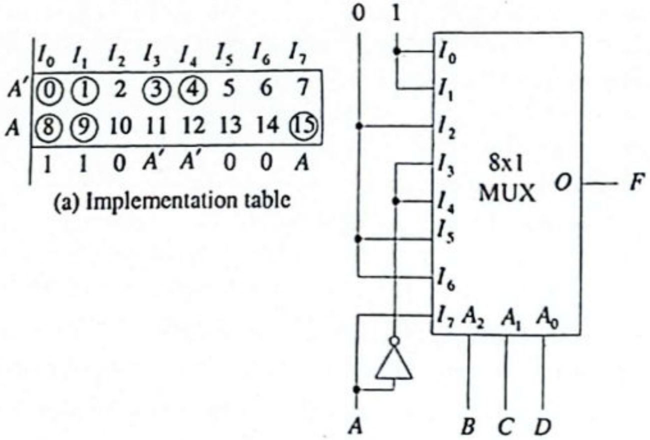
List the inputs of the multiplexer and under them list all the minterms in two rows. The first row lists all those minterms where A is complemented, and the second row all the minterms with A uncomplemented, as shown in Figure 8.35(c). Circle all the minterms of the function and inspect each column separately. If the two minterms in a column are not circled, apply 0 to the corresponding multiplexer input. If the two minterms are circled, apply 1 to the corresponding multiplexer input. If the bottom minterm is circled and the top is not circled, apply A to the corresponding multiplexer input. If the top minterm is circled and the bottom is not circled, apply A' to the corresponding multiplexer input. This procedure follows the condition established during the previous analysis.

Figure 8.35(c) shows the implementation table for the Boolean function $F(A,B,C)=\sum(1,3,5,6)$. From which we obtain the multiplexer connections of Figure 8.35(a). Note that B must be connected to A_1 and C to A_0.

The procedure is illustrated with another example. Consider the function

$$F(A,B,C,D)=\sum(0,1,3,4,8,9,15)$$

This is a four-variable function and therefore we need a multiplexer with three address lines and eight inputs. We apply variables B, C, and D to the address lines. The implementation table is shown in Figure 8.36(a). The first half of the minterms are associated with A' and the second half with A. By circling the minterms of the function and applying the rules for finding values for the multiplexer inputs, we obtain the implementation as shown in Figure 8.36(b).



	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
A'	⓪	①	2	③	④	5	6	7
A	⑧	⑨	10	11	12	13	14	⑮
	1	1	0	A'	A'	0	0	A

(a) Implementation table

(b) Implementation with multiplexer

Figure 8.36 Implementation of the function $F(A,B,C,D)=\sum(0,1,3,4,8,9,15)$ with multiplexer.

Let us now compare the multiplexer method with the decoder method for implementing combinational circuits. The decoder method requires an external gate for each output function, but only one decoder is needed to generate all output functions. The multiplexer method uses smaller-size units but requires one multiplexer for each output function. It would seem reasonable to assume that combinational circuits with a small number of outputs should be implemented with multiplexers. Combinational circuits with many output functions would probably use fewer ICs with the decoder method.

8.8 Demultiplexers

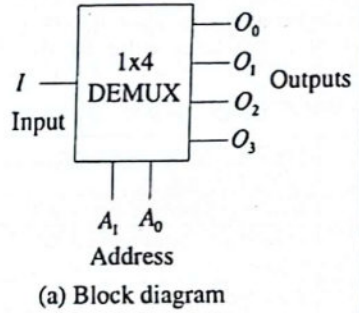
8.8.1 Design of Demultiplexers

A demultiplexer (DEMUX in short) is a combinational circuit that receives information on a single line and transmits this information on one of 2^n possible output lines. The selection of a specific output line is controlled by n address lines.

The block diagram and truth table of a 1-to-4-line demultiplexer is shown in Figure 8.37. From the truth table of Figure 8.37(b), the Boolean functions for the outputs can be written as

$$O_0 = IA_1'A_0'$$
$$O_1 = IA_1'A_0$$
$$O_2 = IA_1A_0'$$
$$O_3 = IA_1A_0$$

The logic implementation of a 1-to-4-line demultiplexer is shown in Figure 8.38.



(a) Block diagram

I	A_1	A_0	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

(b) Truth table

Figure 8.37 Block diagram and truth table of 1-to-4-line demultiplexer.

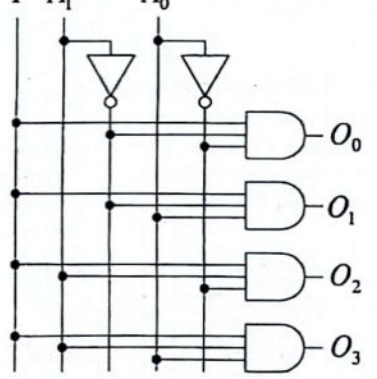


Figure 8.38 Logic implementation for 1-to-4-line demultiplexer.

Note that the logic circuit of Figure 8.38 is indeed a 2-to-4-line decoder with active-HIGH output and active-HIGH enable, where the input line I is the enable input line. Also note that the logic circuit of a 2-to-4-line decoder with active-LOW output and active-LOW enable input, as shown in Figure 8.9, can be used as an 1-to-4-line demultiplexer if the enable input E is taken as the input line. This can be verified from the truth table of Figure 8.8(b) that the E input appears at the output selected by the address line A_1A_0. The only difference is that the values of the unselected lines remain 1.

8.8.2 Available MSI Demultiplexer ICs

Because decoder and demultiplexer operations are obtained from the same circuit, some MSI ICs are referred to as *decoder/demultiplexer*. List of available MSI demultiplexer ICs are given in Table 8.5. Note that the same ICs are also listed in Table 8.2 as decoders.

Table 8.5 Available MSI demultiplexer ICs.

IC No	Description	Output
74139	Dual 1×4 Demultiplexer	Same as input
74155	Dual 1×4 Demultiplexer	Inverted input/Same as input
74156	Dual 1×4 Demultiplexer	Inverted input/Same as input (Open-collector)
74138	1×8 Demultiplexer	Inverted input/Same as input
74154	1×16 Demultiplexer	Same as input

8.8.3 Expansion of Demultiplexer

As in decoders, to meet the larger output needs, demultiplexers can be expanded as shown in Figure 8.39, which shows an 1-to-16-line demultiplexer using 1-to-4-line demultiplexer.

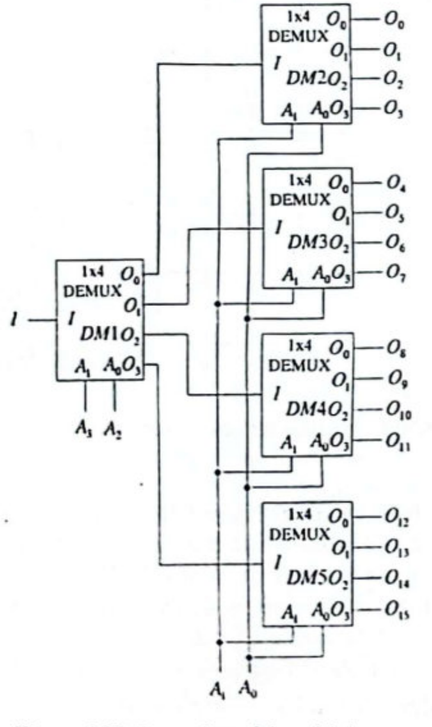


Figure 8.39 Expansion of demultiplexers.

In Figure 8.39, when high-order address bits $A_3A_2 = 00$, the external input I appears at the O_0 output of the demultiplexer *DM*1, which is the input to the demultiplexer *DM*2. Then this input will appear at the output of the demultiplexer *DM*2 which is selected by the low-order address bits A_1A_0. When $A_3A_2 = 01$, the external input I will appear at the O_1 output of the demultiplexer *DM*1, which will then appear at the output of the demultiplexer *DM*3 selected by A_1A_0. When $A_3A_2 = 10$, the external input I will appear at the O_2 output of the demultiplexer *DM*1, which will then appear at the output of the demultiplexer *DM*4 selected by A_1A_0. Finally, when $A_3A_2 = 11$, the external input I will appear at the O_3 output of the demultiplexer *DM*1, which will then appear at the output of the demultiplexer *DM*5 selected by A_1A_0. For example, when $A_3A_2A_1A_0 = 0110$, that is $A_3A_2 = 01$ and $A_1A_0 = 10$, the external input I will appear at the O_1 output of the demultiplexer *DM*1 as $A_3A_2 = 01$. This output is connected to the input of the demultiplexer *DM*3. Then this input to the demultiplexer *DM*3 will appear at its O_2 output as $A_1A_0 = 10$. This output is the output O_6 of the combined demultiplexer.

8.8.4 Combinational Logic Implementation with Demultiplexers

When the input $I = 1$, a demultiplexer functions as a decoder with active-HIGH output. Then combinational logic implementation using demultiplexers becomes similar to combinational logic implementation using decoders. Implementation of the following 3-variable functions using a 1-to-8-line demultiplexer is shown in Figure 8.40.

$$F_1(A,B,C) = \sum(1,2,4,7)$$
$$F_2(A,B,C) = \sum(0,5,6,7)$$

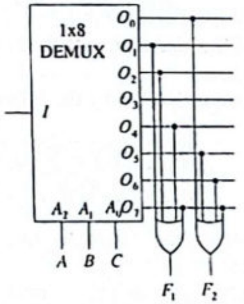


Figure 8.40 Implementation of the functions $F_1(A,B,C) = \sum(1,2,4,7)$ and $F_2(A,B,C) = \sum(0,5,6,7)$ using demultiplexer.

Exercises

8.1 Design the following encoders for both active-LOW and active-HIGH inputs.
  (a) Decimal-to-BCD encoder.
  (b) Decimal-to-Excess-3 encoder.
  (c) Hexadecimal-to-binary encoder.
  (d) 12-4-line encoder.

8.2 Repeat Problem 8.1 for inverted output.

8.3 Design the following decoders for both active-LOW and active-HIGH output.
  (a) 4-to-16-line decoder.
  (b) 4-to-12-line decoder.

8.4 Repeat Problem 8.3 with active-LOW enable input.

8.5 (a) Design a BCD-to-decimal decoder with active-HIGH output.
  (b) Design an Excess-3-to-decimal decoder with both active-LOW and active-HIGH output.

8.6 Repeat Problem 8.5 with active-LOW enable input.

8.7 Design a 1-of-40-line decoder with active-LOW output using BCD-to-decimal decoders with active-LOW output.

8.8 Construct a 5×32 decoder with active-LOW output using four 3×8 decoders with active-LOW output and a 2×4 decoder with active-LOW output.

8.9 Implement a full-adder circuit using decoders with both active-HIGH and active-LOW output.

8.10 Implement a full-subtractor circuit using decoders with both active-HIGH and active-LOW output.

8.11 Design a BCD-to-Excess-3 code converter using a BCD-to-decimal decoder with active-HIGH output and four OR gates.

8.12 A combinational circuit is defined by the following three functions:

*

$$F_1 = A'B' + ABC'$$
$$F_2 = A' + B$$
$$F_3 = AB + A'B'$$

Design the circuit with a decoder with both active-LOW and active-HIGH output and external gates.

8.13 A combinational circuit is defined by the following two functions.

$$F_1(A,B) = \sum(0,3)$$
$$F_2(A,B) = \sum(1,2,3)$$

Implement the combinational circuit by means of a decoder with active-HIGH output and external NAND gate.

8.14 Design an Excess-3-to-BCD code converter using classical method.

8.15 Design an Excess-3-to-BCD code converter using 4-bit binary adder.

8.16 Design a BCD-to-7-segment display decoder with active-HIGH output.

8.17 Design a BCD-to-7-segment display decoder with both active-LOW and active-HIGH output using BCD-to-decimal decoder with active-LOW and active-HIGH output respectively and external gates.

8.18 Design a Hexadecimal-to-7-segment display decoder with both active-LOW and active-HIGH output.

8.19 Design a BCD-to-7-segment display decoder with active-LOW output using 1-to-16-line decoder with active-LOW output and external gates.

8.20 Design a BCD-to-7-segment display decoder with active-LOW output using a BCD-to-decimal decoder with active-LOW output and external gates.

8.21 Draw a logic diagram for odd parity generator/checker for 3-bit message.

8.22 Design an even parity generator/checker for n-bit message.

8.23 Construct a 4-to-1-line multiplexer using a 2-to-4-line decoder with active-HIGH output and external AND and OR gates.

8.24 Design a 4-to-1-line multiplexer with active-HIGH enable input.

8.25 Construct an 8×1 multiplexer with two 4×1 multiplexers having active-LOW enable inputs.

8.26 Construct a 32×1 multiplexer with four 8×1 multiplexers and one 4×1 multiplexer.

8.27 Construct a 32×1 multiplexer using two 16×1 multiplexers.

8.28 Implement the Boolean function

$$F(A,B,C,D) = \sum(0,1,3,4,8,9,15)$$

using a 16×1 multiplexer.

8.29 Implement the following functions with two 4×1 multiplexers, an OR gate, and inverter.

$$F_1 = x'y' + xyz'$$
$$F_2 = x' + y$$
$$F_3 = xy + x'y'$$

8.30 Realize the function

*

$$F(A,B,C,D) = \sum(2,4,6,7,9,10,11,12,15)$$

using
  (a) a 16×1 multiplexer
  (b) an 8×1 multiplexer.

8.31 Implement a full-adder circuit with multiplexer.

8.32 Design a BCD-to-7-segment display decoder with active-LOW output using 4×1 multiplexers and external gates.

8.33 Design a 40×1 multiplexer using 8×1 multiplexers.

8.34 Construct an 1-to-32-line demultiplexer with four 1-to-8-line demultiplexers and one 1-to-4-line demultiplexers.

8.35 Design an 1× 40 demultiplexer using 1×8 demultiplexers.

Further Reading

1. *The TTL Data Book*. Dallas, Texas Instruments, 1988.
2. Richard S. Sandige, *Modern Digital Design*, McGraw-Hill Publishing Company, 1990.
3. R. P. Jain, *Modern Digital Electronics*, 2nd ed., Tata McGraw-Hill Publishing Company Limited, India, 1998.
4. *Microprocessor Data Hand Book*, Revised Ed. BPB Publications, India, 2000.
5. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
6. John P. Uyemura, *Digital Systems Design: An Integrated Approach*, Brooks/Cole Publishing Company, 2000.
7. D. C. Green, *Digital Electronics*, 5th ed., Pearson Education, 2001.
8. M. Morris Mano, *Digital Design*, 3rd ed., Pearson Education, 2002.

CHAPTER 9
FLIP-FLOPS

9.1 Introduction

The digital circuits considered thus far have been combinational, i.e., the outputs at any instant of time are entirely dependent upon the inputs present at that time. Although every digital system is likely to have combinational circuits, most systems encountered in practice also include memory elements, which require that the system be described in terms of *sequential logic*.

A block diagram of a sequential circuit is shown in Figure 9.1. It consists of combinational circuits that accept external inputs and inputs from outputs of memory elements and generate external outputs and inputs to memory elements referred to as excitation.

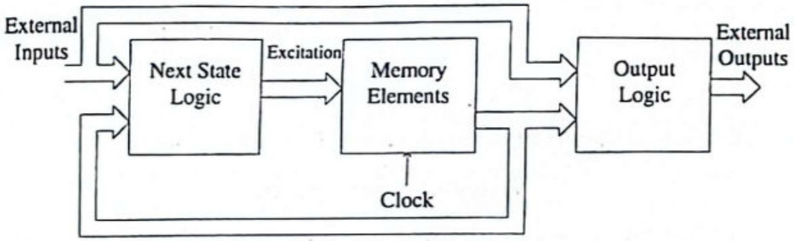


Figure 9.1 Block diagram of a sequential circuit.

A memory element is a device in which one bit of information (1 or 0) can be stored or retained until necessary, and thereafter its content can be replaced by a new value. The contents of memory elements in Figure 9.1 can be changed by the outputs of the combinational circuit which are connected to its input.

The combinational circuit performs certain operations, some of which are used to determine the value to be stored in memory elements. The other operations are performed on external inputs and memory outputs to generate the external outputs.

The above process demonstrates the dependence of the external outputs of a sequential circuit on the external inputs and the present contents of the memory elements (referred to as the *present state* of the sequential circuit). The new contents of the memory elements (referred to as the *next state* of the sequential circuit) depend on the external inputs and the present state. Hence, the output of a sequential circuit is a function of the time sequence of inputs and the *internal states*.

Sequential circuits are classified into two main categories, known as *asynchronous* and *synchronous* sequential circuits depending on timing of their signals.

A sequential circuit whose behavior depends upon the sequence in which the input signals change is referred to as an *asynchronous* sequential circuit. The outputs will be affected whenever the inputs change. The commonly used memory elements in these circuits are time delay devices. These can be regarded as combinational circuits with feedback. Because of the feedback among logic gates, an asynchronous circuit may, at times, become unstable. The instability problem imposes many difficulties on the designer. Hence they are not as commonly used as synchronous sequential circuits.

A sequential circuit whose behavior can be defined from the knowledge of its signal at discrete instants of time is referred to as a *synchronous* sequential circuit. In these circuits, the memory elements are affected only at discrete instants of time. The synchronization is achieved by a timing device known as a *system clock* which generates a periodic train of clock pulses as shown in Figure 9.2. The outputs are affected only with the application of a clock pulse. Synchronous circuits have gained considerable domination and wide popularity and are also known as *clocked-sequential circuits*. They do not manifest instability problems. The memory elements used in clocked-sequential circuits are flip-flops that are capable of storing binary information.

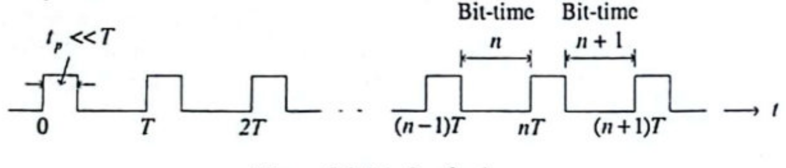


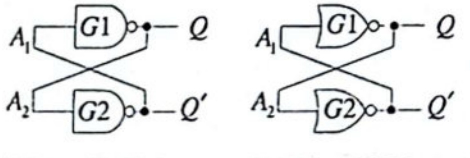
Figure 9.2 A train of pulses.

In this chapter we examine the various types of flip-flops and define their logical properties.

9.2 A 1-Bit Memory Cell

The basic digital memory circuit is known as flip-flop. It has two stable states that are known as the 1 state and the 0 state. It can be obtained by using NAND or NOR gates.

We will be systematically developing a flip-flop circuit starting from the fundamental circuit shown in Figure 9.3. It consists of two inverters $G1$ and $G2$ (NAND and NOR gates used as inverters). The output of $G1$ is connected to the input of $G2$ (A_2) and the output of $G2$ is connected to the input of $G1$ (A_1).



(a) Using NAND gates (b) Using NOR gates

Figure 9.3 Cross-coupled inverters as a memory element.

Let us assume the output of $G1$ to be $Q = 1$, which is also the input of $G2$ ($A_2 = 1$). Therefore, the output of $G2$ will be $Q' = 0$, which makes $A_1 = 0$ and

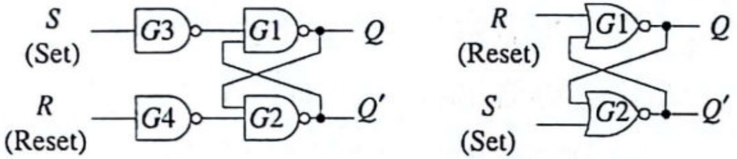
consequently $Q = 1$ which confirms our assumption. In a similar manner, it can be demonstrated that if $Q = 0$, then $Q' = 1$ and this is also consistent with the circuit connections. From the above discussion we note the following:

1. The outputs Q and Q' are always complementary.
2. The circuit has two stable states; in one of the stable state $Q = 1$, which is referred to as the 1 state (or *set state*), whereas in the other stable state $Q = 0$, which is referred to as the 0 state (or *reset state*).
3. If the circuit is in 1 state, it continues to remain in this state and similarly if it is in 0 state, it continues to remain in this state. This property of the circuit is referred to as memory, i.e., it can store 1-bit of digital information.

Since this information is locked or latched in this circuit, this circuit is also referred to as a *latch*.

S-R Latch

In the latch of Figure 9.3, there is no way of entering the desired digital information to be stored in it. In fact, when the power is switched on, the circuit switches to one of the stable states ($Q = 1$ or $Q = 0$) and it is not possible to predict the state. If we replace the inverters $G1$ and $G2$ with 2-input NAND gates as in Figure 9.4(a) and with 2-input NOR gates as in Figure 9.4(b), the other input terminals of the gates can be used to enter the desired digital information. Two additional inverters $G3$ and $G4$ have been added in Figure 9.4(a) for reasons that will become clear from the following discussion.



(a) Using NAND gates (b) Using NOR gates

Figure 9.4 The memory cell with provision for entering data.

If $S = R = 0$, the circuit are exactly the same as that of Figure 9.3. If $S = 1$ and $R = 0$, in Figure 9.4(a), the outputs of $G3$ will be 0 and the output of $G4$ will be 1. Since one of the inputs of the NAND gate $G1$ is 0, its output will certainly be 1. Consequently, both the inputs of the NAND gate $G2$ will be 1 giving an output $Q' = 0$. In Figure 9.4(b), since one of the inputs of the NOR gate $G2$ is 1, its output will certainly be 0. Consequently, both the inputs of the NOR gate $G1$ will be 0 giving an output $Q = 1$. Hence, for this input condition, $Q = 1$ and $Q' = 0$.

Similarly, if $S = 0$ and $R = 1$ then outputs will be $Q = 0$ and $Q' = 1$. The first of these two input conditions ($S = 1$ and $R = 0$) makes $Q = 1$, which is referred to as the *set state*, whereas the second input condition ($S = 0$ and $R = 1$) makes $Q = 0$, which is referred to as the *reset state* or *clear state*. This gives us the means for entering the desired bit in the latch.

Now we see what happens if the input conditions are changed from $S = 1$ and $R = 0$ to $S = R = 0$ or from $S = 0$ and $R = 1$ to $S = R = 0$. The output remains unaltered. This shows the basic difference between a combinational circuit and a sequential circuit, even though the sequential circuit is made up of combinational circuits.

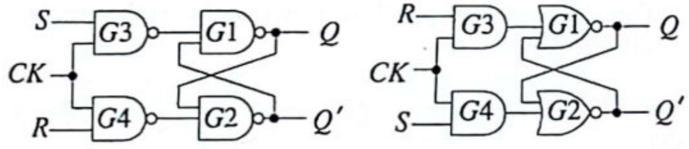
The two input terminals are designated as set (S) and reset (R) because $S = 1$ brings the circuit in set state and $R = 1$ brings it to reset or clear state.

If $S = R = 1$, in Figure 9.4(a) both the outputs Q and Q' will try to become 1 and in Figure 9.4(b) both the outputs Q and Q' will try to become 0, which is not allowed and, therefore, this input condition is prohibited.

9.3 S-R Flip-Flop

9.3.1 Basic S-R Flip-Flop

It is often required to set or reset the memory cell of Figure 9.4 in synchronization with a train of pulses (Figure 9.2) known as clock (abbreviated as *CK*). Such circuit is shown in Figure 9.5 and is referred to as a clocked *set-reset (S-R) flip-flop*.

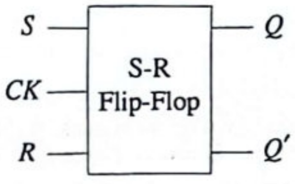


(a) Using NAND latch (b) Using NOR latch

Figure 9.5 Logic diagram of S-R Flip-Flop.

In this circuit, if a clock pulse is present ($CK = 1$), its operation is exactly the same as that of Figure 9.4. On the other hand, when the clock pulse is not present ($CK = 0$), in Figure 9.5(a), the outputs of gates $G3$ and $G4$ are 1 irrespective of the values of S or R and the outputs of gates $G1$ and $G2$ are unchanged. When $CK = 0$, in Figure 9.5(b), the outputs of gates $G3$ and $G4$ are 0 irrespective of the values of S or R and the outputs of gates $G1$ and $G2$ are unchanged. In general, the circuits of Figure 9.5 respond to the inputs S and R only when the clock is present.

The logic symbol of S-R flip-flop is shown in Figure 9.6(a). Assuming that the inputs do not change during the presence of the clock pulse, we can express the operation of the S-R flip-flop as shown in the table of Figure 9.6(b), which is known as characteristic table (also known as truth table). Here S_n and R_n denote the inputs and Q_n the output during the bit time n (Figure 9.2). Q_{n+1} denotes the output after the pulse passes, i.e., in the bit time $n+1$. Figure 9.6(c) shows the reduced characteristic table of S-R flip-flop derived from Figure 9.6(b).



(a) Logic symbol

S_n	R_n	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	?
1	1	1	?

(b) Characteristic table

S_n	R_n	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	?

(c) Reduced characteristic table

Figure 9.6 Logic symbol and characteristic table of S-R flip-flop.

If $S_n = R_n = 0$, and the clock pulse is applied, the output at the end of the clock pulse is same as the output before the clock pulse, i.e., $Q_{n+1} = Q_n$. This is indicated in the first and second rows of Figure 9.6(b) and in the first row of Figure 9.6(c).

If $S_n = 0$ and $R_n = 1$, the output at the end of the clock pulse will be 0, i.e., $Q_{n+1} = 0$. This is indicated in the third and fourth rows of Figure 9.6(b) and in the second row of Figure 9.6(c).

If $S_n = 1$ and $R_n = 0$, the output at the end of the clock pulse will be 1, i.e., $Q_{n+1} = 1$. This is indicated in the fifth and sixth rows of Figure 9.6(b) and in the third row of Figure 9.6(c).

In the circuit of Figure 9.4, it was mentioned that $S = R = 1$ is prohibited. Let us see what happens in the S-R flip-flop of Figure 9.5 if $S_n = R_n = 1$..In Figure 9.5(a), when the clock is present the outputs of gates $G3$ and $G4$ are both 0, making one of the inputs of $G1$ and $G2$ NAND gates 0. Consequently, Q and Q' both will be 1, which is inconsistent with our assumption of complementary outputs. Now, when the clock pulse has passed away ($CK = 0$), the outputs of $G3$ and $G4$ will rise from 0 to 1. Depending upon the propagation delays of the gates, either the stable states $Q_{n+1} = 1$ ($Q'_{n+1} = 0$) or $Q_{n+1} = 0$ ($Q'_{n+1} = 1$) will result. In Figure 9.5(b), when the clock is present the outputs of gates $G3$ and $G4$ are both 1, making one of the inputs of $G1$ and $G2$ NOR gates 1. Consequently, Q and Q' both will be 0, which is also inconsistent with the assumption of complementary outputs. Now, when the clock pulse has passed away ($CK = 0$), the outputs of $G3$ and $G4$ will go from 1 to 0. Depending upon the propagation delays of the gates, either the stable states $Q_{n+1} = 1$ ($Q'_{n+1} = 0$) or $Q_{n+1} = 0$ ($Q'_{n+1} = 1$) will result. That means, in both the Figure 9.5(a) and (b), when $S_n = R_n = 1$ the state of the circuit is undefined, indeterminate or ambiguous and therefore is indicated by a question mark in the seventh and eighth rows of Figure 9.6(b) and in the fourth row of Figure 9.6(c). Thus, the condition $S_n = R_n = 1$ is forbidden and it must not be allowed to occur.

The characteristic table of Figure 9.6(b) is shown in the K-map of Figure 9.7, which gives

$$Q_{n+1} = S_n + R'_n Q_n$$
$$S_n R_n = 0$$

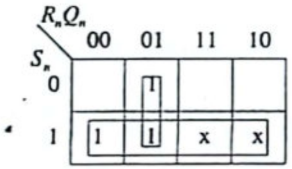


Figure 9.7 Characteristic equation of S-R flip-flop.

The first equation is known as the characteristic equation of S-R flip-flop. This equation specifies the value of the next state as a function of the present state and the inputs. The characteristic equation is an algebraic expression for the binary information of the characteristic table. The two indeterminate states are marked by x's in the K-map, since they may result in either a 1 or a 0. However, the relation $S_n R_n = 0$ must be included as part of the characteristic equation to specify that both S_n and R_n cannot equal 1 simultaneously.

In the design of sequential circuits, we usually come across situations in which the *present state* and the *next state* of the circuit are specified, and we have to find the input conditions that must prevail to cause the desired transition of the state. By the present state and the next state we mean the state of the circuit prior to and after the clock pulse respectively. For example, the output of an S-R flip-flop before the clock pulse $Q_n = 0$ and it is desired that the output does not change when the clock pulse is applied. What input conditions (S_n and R_n values) must exist to achieve this? From the characteristic table of Figure 9.6(b) we obtain the following conditions:

1. $S_n = R_n = 0$ (first row)
2. $S_n = 0$ and $R_n = 1$ (third row)

We conclude from the above conditions that the S_n input must be 0, whereas the R_n input may be either 0 or 1 (don't care). For finding input conditions for all possible situations, the characteristic table of Figure 9.6(b) is rearranged as in Table 9.1, from which the excitation table for S-R flip-flop of Table 9.2 is derived.

Table 9.1 Rearranged characteristic table of S-R flip-flop.

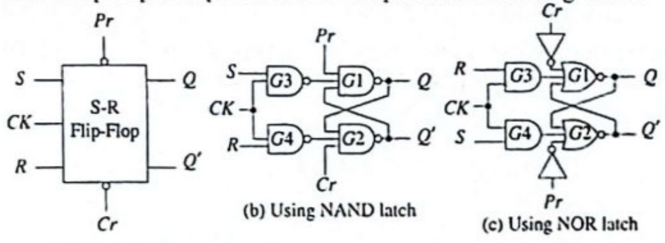
S_n	R_n	Q_n	Q_{n+1}
0	0	0	0
0	1	0	0
1	0	0	1
0	1	1	0
0	0	1	1
1	0	1	1
1	1	0	?
1	1	1	?

Table 9.2 Excitation table of S-R flip-flop.

Q_n	Q_{n+1}	S_n	R_n
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

9.3.2 S-R Flip-Flop with Preset and Clear Inputs

In the S-R flip-flop of Figure 9.5, when the power is switched on, the state of the circuit is uncertain. It may come to set ($Q = 1$) or reset ($Q = 0$) state. In many applications it is desired to initially set or reset the flip-flop, i.e., the initial sate of the flip-flop is to be assigned. This is accomplished by using the direct, or *asynchronous inputs*, referred to as *preset* (*Pr*) and *clear* (*Cr*) inputs. These inputs may be applied at any time between clock pulses and are not in synchronism with the clock. S-R flip-flop with preset and clear inputs is shown in Figure 9.8.



(a) Logic symbol (b) Using NAND latch (c) Using NOR latch

Figure 9.8 S-R flip-flop with preset and clear inputs.

If $Pr = Cr = 1$, the circuits operate in accordance with the characteristic table of S-R flip-flop given in Figure 9.6(b). If $Pr = 0$ and $Cr = 1$, in Figure 9.8(b), the output of $G1$ (Q) will certainly be 1. Consequently, all the three inputs to $G2$ will be 1, which will make $Q' = 0$. In Figure 9.8(c), the output of $G2$ (Q') will certainly be 0. Consequently, all the three inputs to $G1$ will be 0, which will make $Q = 1$. Hence, making $Pr = 0$ sets the S-R flip-flop.

Similarly, if $Pr = 1$ and $Cr = 0$, in Figure 9.8(b), the output of $G2$ (Q') will certainly be 1. Consequently, all the three inputs to $G1$ will be 1, which will make $Q = 0$. In Figure 9.8(c), the output of $G1$ (Q) will certainly be 0. Consequently, all the three inputs to $G2$ will be 0, which will make $Q' = 1$. Hence, making $Cr = 0$ resets the S-R flip-flop.

Once the state of the flip-flop is established asynchronously, the asynchronous inputs Pr and Cr must be connected to 1 before the next clock is applied. The condition $Pr = Cr = 0$ must not be used, since this leads to an uncertain state.

The logic symbol of S-R flip-flop with preset and clear inputs is shown in Figure 9.8(a). In the logic symbol bubbles are used for Pr and Cr inputs, which means these are active-LOW, i.e., the intended function is performed when the signal applied to Pr or Cr is LOW. The operation of S-R flip-flop is summarized in Table 9.3.

Table 9.3 Summary of operation of S-R flip-flop.

Inputs CK Cr Pr	Output Q	Operation performed
1 1 1	Q_{n+1} (Figure 9.6(c))	Normal flip-flop
0 0 1	0	Clear
0 1 0	1	Preset

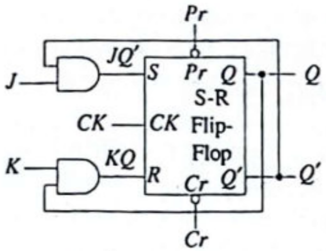
9.4 J-K Flip-Flop

9.4.1 Basic J-K Flip-Flop

The uncertainty in the state of S-R flip-flop when $S_n = R_n = 1$ (seventh and eighth rows of Figure 9.6(b)) can be eliminated by converting it into a J-K flip-flop. The data inputs are J and K, which are ANDed with Q' and Q, respectively, to obtain S and R inputs, i.e.,

$$S = J \cdot Q'$$
$$R = K \cdot Q$$

A J-K flip-flop thus obtained is shown in Figure 9.9(a) and its truth table is given in Figure 9.9(b).



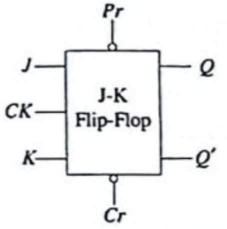
(a) Circuit connection

J_n	K_n	Q_n	S_n	R_n	Q_{n+1}
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	0
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	0	0	1
1	1	0	1	0	1
1	1	1	0	1	0

(b) Truth table

Figure 9.9 An S-R flip-flop converted into J-K flip-flop.

The logic symbol of a J-K flip-flop is shown in Figure 9.10(a). The characteristic table of a J-K flip-flop is shown in Figure 9.10(b) and a reduced characteristic table is also shown in Figure 9.10(c) for convenience.



(a) Logic symbol

J_n	K_n	Q_n	Q_{n+1}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

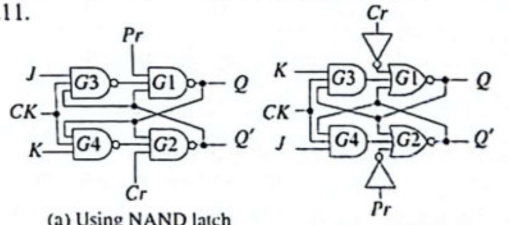
(b) Characteristic table

J_n	K_n	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	Q'_n

(c) Reduced characteristic table

Figure 9.10 Logic symbol and characteristic table of J-K flip-flop.

It is not necessary to use the AND gates of Figure 9.9(a), since the same function can be performed by adding an extra input terminal to each NAND gate $G3$ and $G4$ of Figure 9.8(b) or to each AND gate $G3$ and $G4$ of Figure 9.8(c). With this modification incorporated in Figure 9.8(b) and (c), we obtain the J-K flip-flop as shown in Figure 9.11.



(a) Using NAND latch

(b) Using NOR latch

Figure 9.11 J-K flip-flop.

The characteristic table of Figure 9.10(b) is shown in the K-map of Figure 9.12, which gives the following characteristic equation of J-K flip-flop.

$$Q_{n+1} = J_nQ'_n + K'_nQ_n$$

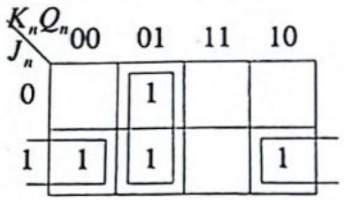


Figure 9.12 Characteristic equation of J-K flip-flop.

The characteristic table of Figure 9.10(b) is rearranged in Table 9.4 for obtaining the excitation table of J-K flip-flop. The excitation table of J-K flip-flop derived from Table 9.4 is given in Table 9.5.

Table 9.4 Rearranged characteristic table of J-K flip-flop for obtaining excitation table.

J_nK_n	Q_nQ_{n+1}
0 0	0 0
0 1	0 0
1 0	0 1
1 1	0 1
0 1	1 0
1 1	1 0
0 0	1 1
1 0	1 1

Table 9.5 Excitation table of J-K flip-flop.

Q_nQ_{n+1}	J_nK_n
0 0	0 x
0 1	1 x
1 0	x 1
1 1	x 0

9.4.2 The Race-Around Condition in J-K Flip-Flop

The difficulty of both inputs 1 ($S = R = 1$) being not allowed in an S-R flip-flop is eliminated in a J-K flip-flop by using the feed-back connection from outputs to the inputs of the gates $G3$ and $G4$ in Figure 9.11. Characteristic table of Figure 9.10(b) assumes that the inputs do not change during the clock pulse ($CK = 1$), which is not true because of the feedback connections. Consider, for example, that the inputs are $J = K = 1$ and $Q = 0$ and a pulse as shown in Figure 9.13 is applied at the clock input. After a time interval Δt equal to the propagation delay through two gates in series (NAND-NAND gates in Figure 9.11(a) and AND-NOR gates in Figure 9.11(b)), the output will change to $Q = 1$ (see seventh row of Figure 9.10(b)). Now we have $J = K = 1$ and $Q = 1$ and after another time interval of Δt the output will change back to $Q = 0$. Hence, we conclude that for the duration t_p of the clock pulse, the output will oscillate back and forth between 0 and 1. At the end of the clock pulse, the value of Q is uncertain. This situation is referred to as the *race-around condition*.

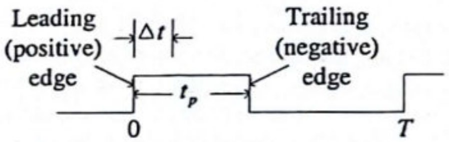


Figure 9.13 A clock pulse.

The race-around condition can be avoided if $t_p < \Delta t < T$. However, it may be difficult to satisfy this inequality because of very small propagation delays in ICs. A more practical method for overcoming this difficulty is the use of the master-slave (M-S) configuration discussed below.

9.4.3 The Master-Slave J-K Flip-Flop

A master-slave J-K flip-flop is a cascade of two S-R flip-flops with feedback from the outputs of the second to the inputs of the first as shown in Figure 9.14. Positive clock pulses are applied to the first flip-flop and the clock pulses are inverted before these are applied to the second flip-flop. (Master-slave J-K flip-flop circuit is presented here with NAND S-R flip-flop. The same circuit with NOR S-R flip-flop is left to the reader as problem.)

When $CK = 1$, the first flip-flop is enabled and the outputs Q_M and Q'_M respond to the inputs J and K according to the table of Figure 9.10(b). At this time, the second flip-flop is inhibited because its clock is LOW ($CK' = 0$). When CK goes LOW ($CK' = 0$), the first flip-flop is inhibited and the second flip-flop is enabled, because now its clock is HIGH ($CK' = 1$). Therefore, the outputs Q_S and Q'_S follow the outputs Q_M and Q'_M, respectively (second and third rows of Figure 9.6(c)). Since the second flip-flop simply follows the first one, it is referred to as the slave and the first one as the master. Hence, this configuration is referred to as *master-slave (M-S) flip-flop*.

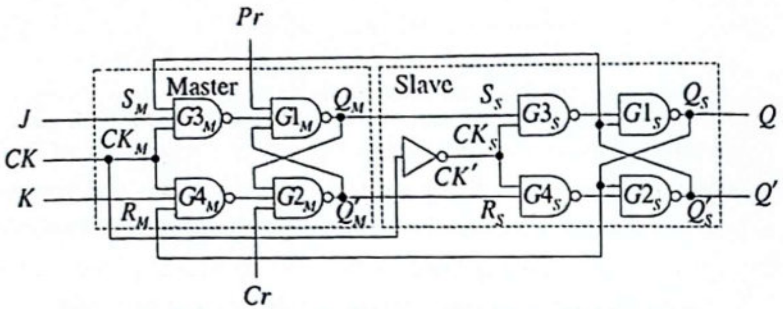


Figure 9.14 Master-slave J-K flip-flop using NAND S-R flip-flop.

In this circuit, the inputs to the gates $G3_M$ and $G4_M$ do not change during the clock pulse, therefore the race-around condition does not exist. The state of the master-slave flip-flop changes at the negative transition (trailing edge) of the clock pulse. The logic symbol of a M-S J-K flip-flop is shown in Figure 9.15. At the clock input terminal, the symbol > is used to illustrate that the output changes when the clock makes a transition and the accompanying bubble signifies negative transition (change in *CK* from 1 to 0).

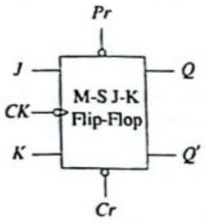
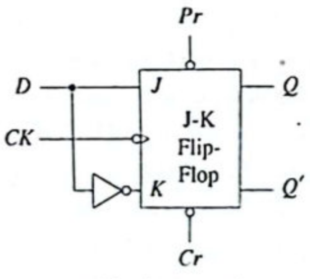


Figure 9.15 Logic symbol of M-S J-K flip-flop.

9.5 D Flip-Flop

If we convert a J-K flip-flop circuit as shown in Figure 9.16(a), we obtain a D flip-flop. The truth table of the circuit is shown in Figure 9.16(b).



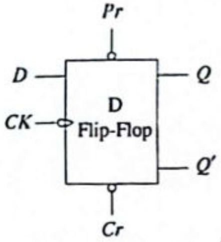
(a) Circuit connection

D_n	Q_n	J_n	K_n	Q_{n+1}
0	0	0	1	0
0	1	0	1	0
1	0	1	0	1
1	1	1	0	1

(b) Truth table

Figure 9.16 A J-K flip-flop converted into a D flip-flop.

The logic symbol of a D flip-flop is shown in Figure 9.17(a) and its characteristic table is shown in Figure 9.17(b). A reduced characteristic table is also shown in Figure 9.17(c) for convenience.



(a) Logic symbol

D_n	Q_n	Q_{n+1}
0	0	0
0	1	0
1	0	1
1	1	1

(b) Characteristic table

D_n	Q_{n+1}
0	0
1	1

(c) Reduced characteristic table

Figure 9.17 Logic symbol and characteristic table of a D flip-flop.

A D flip-flop has only one input referred to as D-input or data-input. From the characteristic table of Figure 9.17(c), it is clear that the output Q_{n+1} at the end of the clock pulse equals the input D_n before the clock pulse. This is equivalent to saying that the input data appears at the output at the end of the clock pulse. Thus, the transfer of data from the input to the output is delayed and hence the name *delay (D) flip-flop*. The D flip-flop is either used as delay device or as a latch to store 1-bit of binary information.

The characteristic table of Figure 9.17(b) is shown in the K-map of Figure 9.18, which gives the following characteristic equation of a D flip-flop

$$Q_{n+1} = D_n$$

$D_n \backslash Q_n$	0	1
0		
1	1	1

Figure 9.18 Characteristic equation of D flip-flop.

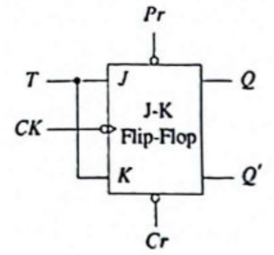
The excitation table of a D flip-flop can be easily derived from the characteristic table of Figure 9.17(b) and is shown in Table 9.6.

Table 9.6 Excitation table of D flip-flop.

$Q_n Q_{n+1}$	D_n
0 0	0
0 1	1
1 0	0
1 1.	1

9.6 T Flip-Flop

If we convert a J-K flip-flop circuit as shown in Figure 9.19(a), we obtain a T flip-flop. The truth table of the circuit is shown in Figure 9.19(b).



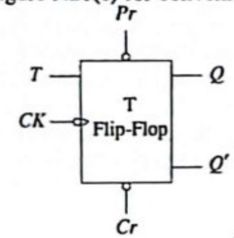
(a) Circuit connection

T_n	Q_n	J_n	K_n	Q_{n+1}
0	0	0	0	0
0	1	0	0	1
1	0	1	1	1
1	1	1	1	0

(b) Truth table

Figure 9.19 A J-K flip-flop converted into a T flip-flop.

The logic symbol of a T flip-flop is shown in Figure 9.20(a) and its characteristic table is shown in Figure 9.20(b). A reduced characteristic table is also shown in Figure 9.20(c) for convenience.



(a) Logic symbol

T_n	Q_n	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

(b) Characteristic table

T_n	Q_{n+1}
0	Q_n
1	Q'_n

(c) Reduced characteristic table

Figure 9.20 Logic symbol and characteristic table of a T flip-flop.

A T flip-flop has only one input, referred to as T-input. From the characteristic table of Figure 9.20(c), it is clear that if $T = 1$ the flip-flop acts as a toggle switch. For every clock pulse, the output Q changes or toggles and hence the name *toggle (T) flip-flop*.

The characteristic table of Figure 9.20(b) is shown in the K-map of Figure 9.21, which gives the following characteristic equation of a T flip-flop

$$Q_{n+1} = T'_n Q_n + T_n Q'_n$$

$T_n \backslash Q_n$	0	1
0		1
1	1	

Figure 9.21 Characteristic equation of a T flip-flop.

The excitation table of a T flip-flop can easily be derived from the characteristic table of Figure 9.20(b) and is shown in Table 9.7.

Table 9.7 Excitation table of T-flip-flop.

$Q_n Q_{n+1}$	T_n
0 0	0
0 1	1
1 0	1
1 1	0

9.7 Conversion From One Type of Flip-Flop to Another Type

In earlier sections, we have discussed conversion from S-R to J-K, J-K to D, and J-K to T flip-flops. Now, we shall discuss the conversion from one type of flip-flop to another type by using a formal technique.

Consider the general model for conversion from one type of flip-flop to another as shown in Figure 9.22. In this, we are required to design the combinational logic for converting new input definition into input requirements for the given flip-flop that will cause the given flip-flop to perform as desired.

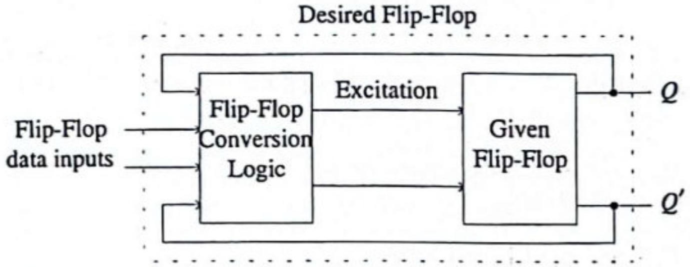


Figure 9.22 The general model used to convert one type of flip-flop to another type.

To design the conversion logic we need to make a truth table with data input(s) and Q as the inputs and the input(s) of the given flip-flop as the output(s). This is done by writing the characteristic table of the desired flip-flop and then determining the excitation requirements for the given flip-flop from excitation table of the given flip-flop. The conventional method of combinational logic design then follows as usual. The conversion of an S-R flip-flop into a J-K flip-flop is illustrated below.

The characteristic table of the desired J-K flip-flop and the excitation table of the given S-R flip-flop are combined in Table 9.8 to form the truth table for converting S-R flip-flop into a J-K flip-flop. The K-maps for S_n and R_n are given in Figure 9.23, which give

$$S_n = J_n Q'_n$$
$$R_n = K_n Q_n.$$

Table 9.8 Truth table for converting an S-R flip-flop into a J-K flip-flop.

Characteristic table of the desired J-K flip-flop		Excitation required for the given S-R flip-flop
J_n K_n Q_n	Q_{n+1}	S_n R_n
0 0 0	0	0 x
0 0 1	1	x 0
0 1 0	0	0 x
0 1 1	0	0 1
1 0 0	1	1 0
1 0 1	1	x 0
1 1 0	1	1 0
1 1 1	0	0 1

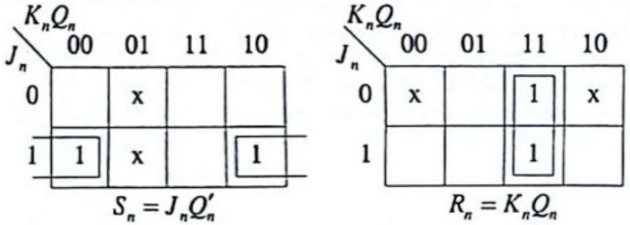


Figure 9.23 K-map for excitation function of S-R flip-flop for converting it into a J-K flip-flop.

Thus, we see that the circuit resulting from this design is the same as that shown in Figure 9.9(a).

9.8 Triggering of Flip-Flops

All flip-flops other than the master-slave type discussed earlier are level triggered, i.e., the outputs respond to the inputs (according to the characteristic table) as long as the clock is present.

The master-slave flip-flops are also referred to as the *pulse-triggered* flip-flop, i.e., the outputs respond to the inputs when a pulse is applied at the clock input. The master responds when the clock is present ($CK = 1$) and the output of the slave will be available at the falling edge of the clock pulse ($CK = 0$). As discussed in Section 9.4, this eliminates the problem of race-around condition. In this the data is *locked-out* at the falling edge of the clock pulse, i.e., the changes occurring at the inputs once *CK* goes to 0 will not affect the operation of the flip-flop.

The inputs to the flip-flop may change during the presence of the clock pulse due to certain operations in the system. This causes uncertainty in the outputs of the flip-flop, which is eliminated by using edge-triggered flip-flops. In the case of an edge-triggered flip-flop, the transfer of information from data input(s) to the output of the flip-flop occurs at the positive (or negative) edge of the clock pulse. The only time the outputs can change state is during the brief interval of time when the clock signal is making a transition from the 0 to 1, or from the 1 to 0 states. A flip-flop that responds only to rising (or falling) edge is referred to as *positive-edge-triggered* (or *negative-edge-triggered*) flip-flop. The data lockout occurs at the end of the edge. The logic symbol used for an edge-triggered flip-flop is the same as that of a master-slave flip-flop.

The block diagram of an edge-triggered flip-flop is shown in Figure 9.24. The edge-detector produces a narrow positive pulse ($CK*$) that occurs coincident with the transition of the *CK* input pulse. The output of the flip-flop responds to the inputs during this narrow pulse.

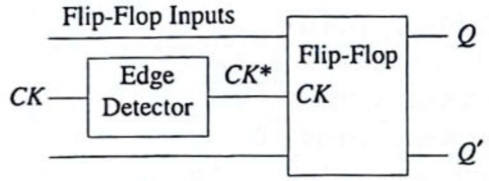
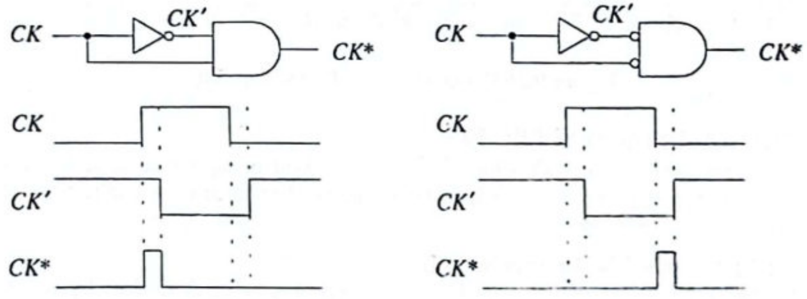


Figure 9.24 Block diagram of an edge-triggered flip-flop.

The generation of *CK** pulse for positive-edge-triggered flip-flop is shown in Figure 9.25(a). The inverter produces a delay of a few nanoseconds so that the transition of CK' occurs a little bit after that of *CK*. The AND gate produces an output pulse that is HIGH only for the few nanoseconds when *CK* and CK' are both HIGH. The result is a narrow pulse at *CK**, which occurs on the positive edge of *CK*. The generation of *CK** pulse for negative-edge-triggered flip-flop is shown in Figure 9.25(b). In this case, the AND gate produces an output pulse that is HIGH only for the few nanoseconds when *CK* and CK' are both LOW. The result is a narrow pulse at *CK**, which occurs on the negative edge of *CK*.



(a) Positive-edge detector

(b) Negative-edge detector

Figure 9.25 Edge detector circuit.

Since the CK^* pulse is HIGH for only a few nanoseconds, the output Q is affected by the flip-flop inputs for a short time during and after the occurrence of the active-edge of the clock. This is what gives the flip-flop its edge-triggered property.

9.9 Available Flip-Flop ICs

Some available flip-flop ICs are listed in Table 9.9. Pin assignment of some of the useful Flip-flops is shown in Figure 9.26.

Table 9.9 Available flip-flop ICs.

IC No.	Description
7473	Dual J-K master-slave flip-flop
7474	Dual +edge-triggered D flip-flop
7475	Quad +level-triggered D flip-flop
7476	Dual J-K master-slave flip-flop
74107	Dual J-K master-slave flip-flop
74109	Dual +edge-triggered J-K′ flip-flop
74112	Dual –edge-triggered J-K flip-flop
74113	Dual –edge-triggered J-K flip-flop
74173	Quad +edge-triggered D flip-flop with 3-state output
74174	Hex +edge-triggered D flip-flop
74175	Quad +edge-triggered D flip-flop

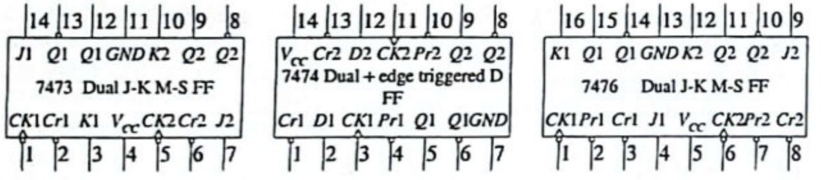


Figure 9.26 Pin assignment of some flip-flops.

9.10 Applications of Flip-Flops

The main application of flip-flops is as memory unit of sequential circuits. This application will be obvious in subsequent chapters. Here three more applications of flip-flops are discussed.

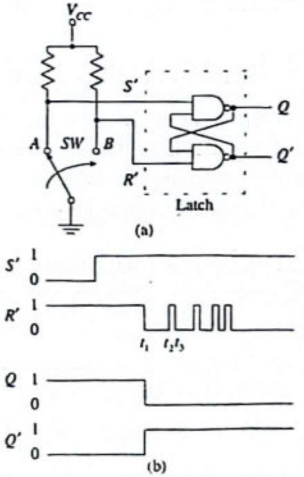
9.10.1 Bounce-Elimination Switch

Mechanical switches are employed in digital systems as input devices by which digital information (0 or 1) is entered into the system. There is a very serious problem associated with these switches, viz., *switch-bouncing* (or *chattering*). When the arm of the switch is thrown from one position to another, it chatters or bounces several times before finally coming to rest in the position of contact. The bounce is the result of the spring-loaded impact of the switch throw contact and the pole contacts.

In a sequential circuit, if a 1 is to be entered through a switch, then the switch is thrown to the corresponding position. As soon as it is thrown to this position, the output is 1 but the output oscillates between 0 and 1 for some time due to make and break (bouncing) of the switch at the point of contact before coming to rest. This changes the output of the sequential circuit and creates difficulties in the operation of the system. This problem is eliminated by using bounce-elimination switches.

The circuit of Figure 9.27(a) acts as a bounce-elimination (chatterless) switch. The wave forms at S', R', Q, and Q' are illustrated in Figure 9.27(b). The switch (SW) is thrown from position A to B at $t = 0$. Therefore, at $t = 0^+$ the voltage at S' (A) will be V_{CC} (logic value 1) and will continue to remain so as long as the switch is not thrown to position A again. At R' (B), the voltage at $t = 0^-$ is V_{CC} (logic value 1) and goes to 0V (logic value 0) at t_1 (t_1 being the time delay of the switch). The switch arm makes contact at B at $t = t_1$, and then bounces off. Therefore, the level at R' changes from 0 to 1 and vice-versa. This is illustrated in Figure 9.27(b). Between $t = 0$ and $t = t_1$, both the inputs S' and R' are at 1 and therefore, Q does not change. The output Q changes at t_1 and becomes 0. Now, even when R' is changing at t_2, t_3, etc. Q does not change. This shows that it is a chatterless switch.

The latch used in Figure 9.27(a) can be replaced by the IC 74279, which is a quad S′-R′ latch.



(a)

(b)

Figure 9.27 (a) A bounce-elimination switch. (b) waveforms of S', R', Q, and Q'.

9.10.2 Random-Access Memory

In computers, digital control systems, information processing systems, etc. it is necessary to *store* digital data and *retrieve* the data as desired. Flip-flops can be used for making memories in which data can be stored for any desired length of time and then read out whenever required. In such a memory, data can be put into (*writing* into memory) or retrieved from (*reading* from the memory) the memory in a random fashion and is known as *random-access memory*.

A 1-bit read/write memory is shown in Figure 9.28, which is the basic memory element and memory ICs are built around a system of basic 1-bit cell. In this memory cell, a level triggered D flip-flop is used which has *Q* output that follows the *D* input as long as *CK* terminal is at 1. The moment the *CK* input changes to 0, the *Q* output does not change and it retains the *D* input level that existed just before the transition from 1 to 0 at input *CK*. This input is used to select the memory cell. In the 1-bit cell shown there are three inputs – D_i (data input), A_n (address select) and R/W' (read/write control) and one output D_o (data output). $A_n = 1$ enables the cell for reading or writing operation, R/W' at 1 is for reading from the cell and 0 for writing into the cell. As long as $A_n = 0$, all input and output activities are blocked, and the cell is in the hold mode where its stored output is protected.

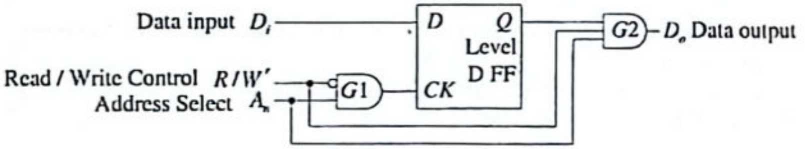


Figure 9.28 A 1-bit read/write memory cell.

The complete function of this cell can be understood from the function table of Table 9.10. The read operation is nondestructive, that is, the stored bit can be read out any number of times without disturbing it. The stored bit will be protected as long as power is on. Therefore, this type of memory is known as *volatile memory*.

Table 9.10 Function table of 1-bit memory cell.

Inputs			Mode
A_n	R/W'	D_i	
0	x	x	Hold, $D_o = 0$.
1	0	0	Write 0 into memory, $D_o = 0$.
1	0	1	Write 1 into memory, $D_o = 0$.
1	1	x	Read, D_o = stored D_i bit.

As far as writing into the cell is concerned, it is not required to be cleared before entering the new bit. Whenever a new bit is entered the earlier one gets destroyed automatically.

9.10.3 Frequency Divider

It is some times needed to generate low-frequency clock signal from a high-frequency clock signal. This can be achieved by the frequency divider circuit shown in Figure 9.29(a). Each T flip-flop has its *T* input connected to 1, so that it will change states (toggles) whenever the *CK* signal goes from 1 to 0. The high-frequency clock signal is applied to the *CK* input of flip-flop *FF*1. Output Q_1 is connected to the *CK* input of flip-flop *FF*2, and output Q_2 is connected to the *CK* input of flip-flop *FF*3. The waveforms of Figure 9.29(b) shows how the flip-flops change states as the clock pulse is applied.

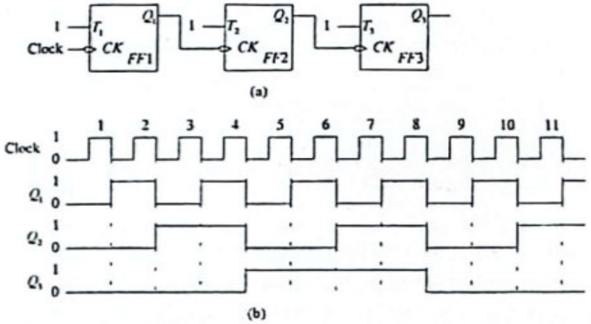


Figure 9.29 Frequency divider circuit using T flip-flops.

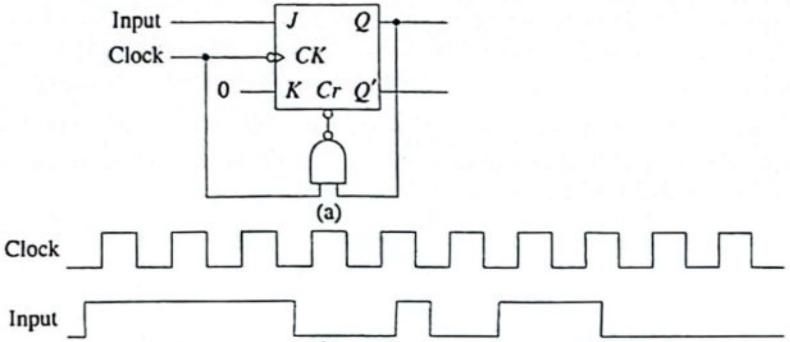
Flip-flop *FF*1 toggles on the negative-edge transition of each input clock pulse. Thus, the Q_1 output waveform has a frequency that is exactly one-half of the input clock frequency. Flip-flop *FF*2 toggles each time Q_1 output goes from 1 to 0. The Q_2 output waveform has a frequency equal to exactly one-half the frequency of the Q_1 output and, therefore, one-fourth of the input clock frequency. Flip-flop *FF*3 toggles each time Q_2 output goes from 1 to 0. Thus, the Q_3 output waveform has one-half the frequency of Q_2 output and, therefore, one-eighth of the input clock frequency.

As described above, each flip-flop divides the frequency of its input by 2. Thus, if we were to add a fourth flip-flop to the chain, it would have a frequency equal to one-sixteenth of the input clock frequency, and so on. Using the appropriate number of flip-flops, this circuit could divide a frequency by any power of 2. More specifically, using *n* flip-flops would produce an output frequency from the last flip-flop that is equal to $1/2^n$ of the input frequency.

Exercises

9.1 Draw circuits for M-S J-K flip-flop with NOR S-R flip-flop.

9.2 For the circuit shown in Figure 9.30(a) the clock and input waveforms shown in Figure 9.30(b) are applied. Sketch the waveforms of Q and Q' if the flip-flop is edge-triggered.



(a)

(b)

Figure 9.30 Figure for Exercise 9.2.

9.3 Repeat Exercise 9.2 if the flip-flop is master-slave.

9.4 If the waveforms shown in Figure 9.31 are applied to the circuit of Figure 9.30(a), sketch the output (Q) waveform assuming M-S flip-flop.

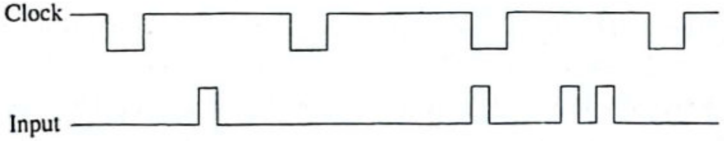


Figure 92.31 Figure for Exercise 9.4.

9.5 Verify that if Q' output of a D flip-flop is connected to D input, it acts as a toggle switch.

9.6 Perform the following flip-flop conversions:

(a) S-R flip-flop into D flip-flop.
(b) S-R flip-flop into T flip-flop.
(c) T flip-flop into D flip-flop.
(d) D flip-flop into T flip-flop.
(e) D flip-flop into J-K flip-flop.
(f) T flip-flop into J-K flip-flop.

9.7 Verify the operation of the debounce switches shown in Figure 9.32.

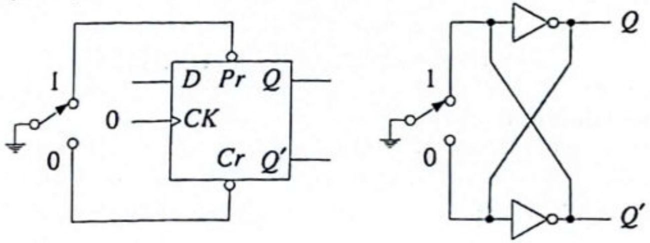


Figure 9.32 Debounce switches for Exercise 9.7.

Further Reading

1. E. J. McCluskey, *Logic Design Pronciples*, Prentice-Hall, Inc., 1986.
2. *The TTL Data Book*. Dallas, Texas Instruments, 1988.
3. Zvi Kohavi, *Switching and Finite Automata Theory*, 2nd ed., McGraw-Hill Higher Education, 1990.
4. Richard S. Sandige, *Modern Digital Design*, McGraw-Hill Publishing Company, 1990.
5. R. P. Jain, *Modern Digital Electronics*, 2nd ed., Tata McGraw-Hill Publishing Company Limited, India, 1998.
6. *Microprocessor Data Hand Book*, Revised Ed. BPB Publications, India, 2000.
7. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
8. John P. Uyemura, *Digital Systems Design: An Integrated Approach*, Brooks/Cole Publishing Company, 2000.
9. D. C. Green, *Digital Electronics*, 5th ed., Pearson Education, 2001.
10. Ronald J. Tocci and Neal S. Widmer, *Digital Systems: Principles and Applications*, 8th ed., Prentice-Hall, Inc., 2001.
11. V. K. Jain, *Switching Theory and Digital Electronics*, Khanna Publishers, India, 2002.
12. M. Morris Mano, *Digital Design*, 3rd ed., Pearson Education, 2002.
13. Samuel C. Lee, *Digital Circuits and Logic Design*, Prentice-Hall, Inc., 2002.

CHAPTER 10
SEQUENTIAL LOGIC DESIGN

10.1 Introduction

Sequential circuit is introduced in Chapter 9. The block diagram of a sequential circuit is shown in Figure 9.1. A logic circuit is recognized as a sequential circuit if it includes flip-flops. The flip-flops may be of any type and the logic circuit may or may not include combinational gates. The external outputs of a sequential circuit are a function of external inputs and the present state of the flip-flops. The next state of the flip-flops is also a function of external inputs and the present state. Thus, a sequential circuit is specified by a time sequence of external inputs, outputs, and internal states.

There are two types of sequential circuits. Their classification depends on the timing of their signals. A *synchronous* sequential circuit is a circuit whose behavior can be defined from the knowledge of its signals at discrete instants of time. The behavior of an *asynchronous* sequential circuit depends upon the order in which its input signals change and can be affected at any instant of time. An asynchronous sequential circuit may, at times, become unstable. The instability problem imposes many difficulties on the designer. Hence they are not as commonly used as synchronous sequential circuits. In synchronous sequential circuits the outputs are affected only with the application of a clock pulse (Figure 9.2). Synchronous sequential circuits have gained considerable domination and wide popularity and are also known as *clocked-sequential-circuits*. They do not manifest instability problems.

The sequential circuits discussed in this chapter and the following chapters are exclusively of synchronous or clocked type. Unless otherwise explicitly specified, in the following texts, the term sequential circuit refers to synchronous or clocked sequential circuit. In this chapter analysis and design of synchronous sequential circuits are discussed.

10.2 Representation of Sequential Circuits

There are three main methods to represent the analytical properties of a sequential circuit:

1. State equations and output equations.
2. Transition table and state table.
3. Transition diagram and State diagram.

These methods of representing a sequential circuit are illustrated with the circuit of Figure 10.1.

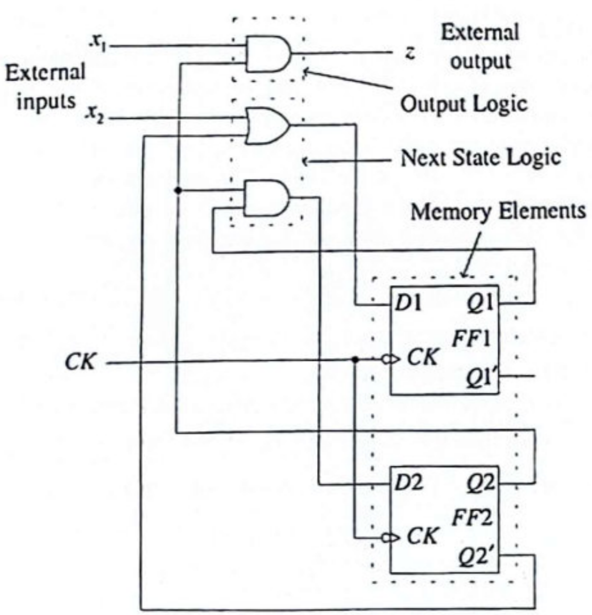


Figure 10.1 An example sequential circuit.

State and Output Equations

The behavior of a sequential circuit is determined by next state equations and output equations. For deriving next state equations, excitation equations are also derived.

The *excitation equations* describe the excitation functions to the flip-flops and are derived from the next state logic of the sequential circuit. For the circuit of Figure 10.1, the excitation equations are

$$D1_n = x_2 + Q2'_n$$
$$D2_n = Q1_n Q2_n$$

The *next state equations* describe the next state function of the flip-flops and are derived from the excitation equations in conjunction with the characteristic equation of the flip-flops. For the circuit of Figure 10.1, the next state equations are

$$Q1_{n+1} = D1_n = x_2 + Q2'_n$$
$$Q2_{n+1} = D2_n = Q1_n Q2_n$$

The *output equations* describe the external output functions of a sequential circuit and are derived from the output logic of the circuit. For the circuit of Figure 10.1, the output equation is

$$z = x_1 Q2_n$$

Transition and State Tables

The *transition table* of a sequential circuit shows the next state and the output for all possible combinations of external input and present state of the circuit. The information is displayed in table of the type shown in Table 10.1. The columns of the table correspond to the possible values of the input variables and the rows correspond to the possible states of the circuit. The entry corresponding to the *i*th row and *j*th column is the next state and the output of the circuit. The next state is derived from the next state equations and the output is derived from the output equations of the circuit.

The transition table for the circuit of Figure 10.1 is shown in Table 10.1. There are four possible combinations of two external input variables x_1 and x_2 and four possible states of the circuit for two flip-flops.

The entries in the table are made as follows: Take for example the second row and third column corresponding to present state 01 and input 11, we have $x_1 = 1$, $x_2 = 1$, $Q1_n = 0$, $Q2_n = 1$. From the next state and output equations, we get

$$z = x_1 Q2_n = 1 \cdot 1 = 1$$

$$Q1_{n+1} = x_2 + Q2'_n = 1 + 0 = 1$$

$$Q2_{n+1} = Q1_n Q2_n = 0 \cdot 1 = 0$$

Therefore, the next state is 10 and output is 1. The table is similarly completed for all input combinations and present states.

Table 10.1 Transition table for the sequential circuit of Figure 10.1.

Present state $Q1_n Q2_n$	Next state $Q1_{n+1} Q2_{n+1}$ Input $x_1 x_2$ 00 01 11 10	Output z Input $x_1 x_2$ 00 01 11 10
00	10 10 10 10	0 0 0 0
01	00 10 10 00	0 0 1 1
11	01 11 11 01	0 0 1 1
10	10 10 10 10	0 0 0 0

Sometimes the states of a sequential circuit are denoted by symbols and the transition table is redrawn in a compact manner as in Table 10.2. Such a table is known as *state table*. Denoting the state 00 by *A*, 01 by *B*, 11 by *C*, and 10 by *D* and redrawing the transition table of Table 10.1, we obtain the state table of Table 10.2 for the circuit of Figure 10.1.

Table 10.2 State table for the sequential circuit of Figure 10.1.

Present state $Q1_n Q2_n$	Next state $Q1_{n+1} Q2_{n+1}$, Output z Input $x_1 x_2$ 00 01 11 10
A	*D*,0 *D*,0 *D*,0 *D*,0
B	*A*,0 *D*,0 *D*,1 *A*,1
C	*B*,0 *C*,0 *C*,1 *B*,1
D	*D*,0 *D*,0 *D*,0 *D*,0

Transition and State Diagrams

Transition diagram and state diagram are graphical means of displaying the information contained in the transition table and state table, respectively. In a *transition diagram* each binary state of the sequential circuit is represented by a node. A transition from one state to another is represented by a directed edge. The edge is labeled with the input combination causing the transition and the corresponding output resulting from the transition. The transition diagram for the sequential circuit of Figure 10.1 is shown in Figure 10.2, which is derived from the transition table of Table 10.1.

To understand the procedure of drawing a transition diagram from the transition table, consider that 00 is the present state. The next state is 10 for input combinations 00, 01, 11, 10 and the output is 0 in all the cases. An edge is drawn from 00 to 10 directing towards 10 and labeled as 00/0, 01/0, 11/0, 10/0. Similarly if 11 is the present state and the input is 00 or 10, then the next state is 01; and input is 01 or 11, then the next state is the same as the present state 11. The transition are, therefore, shown by directed edge from 11 to 01 labeled as 00/0, 10/1 and a directed self-loop at 11 labeled as 01/0, 11/1.

The *state diagram* is similar to transition diagram except that states are represented by symbols. The state diagram for the sequential circuit of Figure 10.1 is shown in Figure 10.3, which is derived from the state table of Table 10.2.

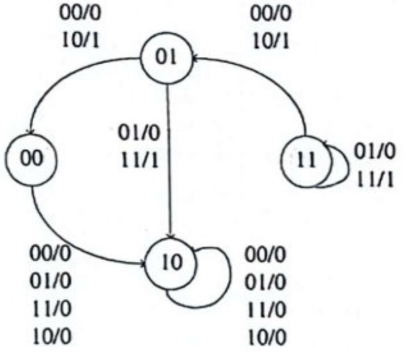


Figure 10.2 Transition diagram for the sequential circuit of Figure 10.1.

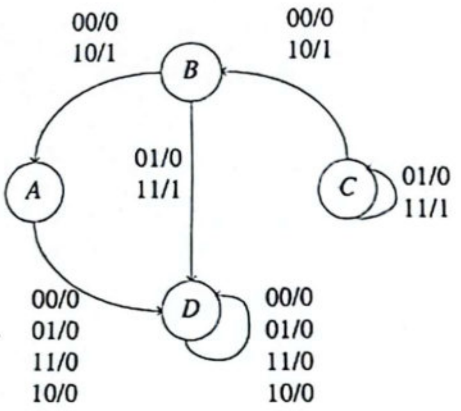


Table 10.3 State diagram for the sequential circuit of Figure 10.1.

10.3 Analysis Procedure of Sequential Circuits

The analysis procedure for sequential circuit starts from the logic diagram and culminates in a transition diagram. Analysis of sequential circuit can be performed through the following steps:

1. Write the excitation equations of the flip-flops and the output equations from the combinational logic circuit.
2. Write the next state equations of the flip-flops from the excitation equations in conjunction with the characteristic equations of the flip-flops.
3. Prepare transition table from the next state equations and the output equations.
4. Draw the transition diagram from the transition table.

The analysis procedure is illustrated using the sequential circuit of Figure 10.4.

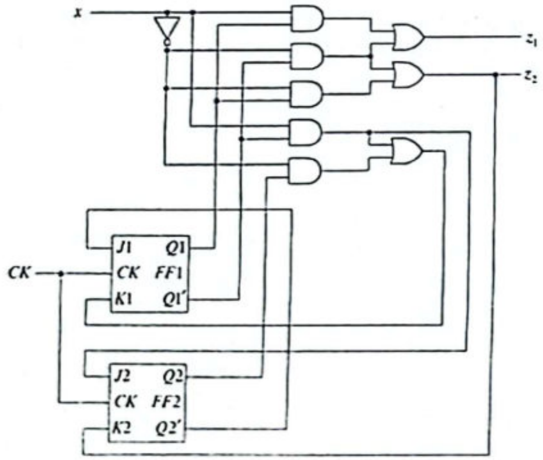


Figure 10.4 An example sequential circuit for analysis.

Step 1. The excitation equations of the flip-flops are written from the combinational circuit as:

$$J1_n = Q2'_n$$
$$K1_n = xQ1'_n + x'Q2_n$$
$$J2_n = xQ1'_n$$
$$K2_n = x'Q1'_n + x'Q1_n = x'$$

The output equations are written from the combinational logic circuits as:

$$z_1 = xQ1_n + x'Q1'_n$$
$$z_2 = x'Q1'_n + x'Q1_n = x'$$

Step 2. The next state equations of the J-K flip-flops are written from the excitation equations in conjunction with characteristic equation of J-K flip-flops as:

$$\begin{aligned}
Q1_{n+1} &= J1_nQ1'_n + K1'_nQ1_n \\
&= Q2'_nQ1'_n + (xQ1'_n + x'Q2_n)'Q1_n \\
&= Q1'_nQ2'_n + (x' + Q1_n)(x + Q2'_n)Q1_n \\
&= Q1'_nQ2'_n + x'Q1_nQ2'_n + xQ1_n + Q1_nQ2'_n \\
&= Q1'_nQ2'_n + Q1_nQ2'_n(x' + 1) + xQ1_n \\
&= Q2'_n(Q1'_n + Q1_n) + xQ1_n \\
&= Q2'_n + xQ1_n \\
Q2_{n+1} &= J2_nQ2'_n + K2'_nQ2_n \\
&= xQ1'_nQ2'_n + xQ2_n
\end{aligned}$$

Step 3. The transition table for the circuit is prepared from the next state equations and output equations and shown in Table 10.3.

Table 10.3 Transition table for the sequential circuit of Figure 10.4.

Present state $Q1_nQ2_n$	Next state $Q1_{n+1}Q2_{n+1}$ Input x		Outputs z_1z_2 Input x	
	0	1	0	1
00	10	11	11	00
01	00	01	11	00
11	00	11	01	10
10	10	10	01	10

Step 4. The transition diagram for the circuit is prepared from the transition table of Table 10.3 and is shown in Figure 10.5.

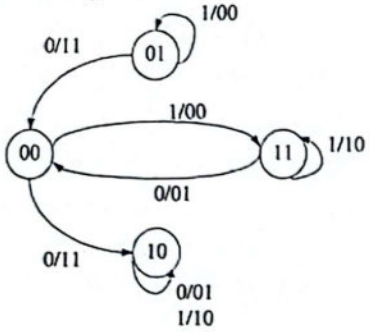


Figure 10.5 Transition diagram for the sequential circuit of Figure 10.4.

10.4 State Reduction and Assignment

The design of a sequential circuit starts from a set of specifications and culminates in a logic diagram. Design procedure for sequential circuits is presented in Section 10.5. This section discusses certain properties of sequential circuits that may be used to reduce the number of gates and flip-flops during the design.

10.4.1 State Reduction

Any design process must consider the problem of minimizing the cost of the final circuit. The two most obvious cost reductions are reductions in the number of flip-flops and the number of gates. The reduction of the number of flip-flops in a sequential circuit is referred to as the *state reduction* problem. State reduction algorithms are concerned with procedures for reducing the number of states in a state table while keeping the external input-output requirements unchanged. Since m flip-flops produce 2^m states, a reduction in the number of states may (or may not) result in a reduction in the number of flip-flops. An unpredictable effect in reducing the number of flip-flops is that sometimes the equivalent circuit (with fewer flip-flops) may require more combinational gates.

The problem of state reduction is to find ways of reducing the number of states in a sequential circuit without altering the input-output relationships. It is more convenient to apply procedures for state reduction in state table than in state diagram. An algorithm for the state reduction of a completely specified state table is given here without proof: *Two states are said to be equivalent if, for each member of the set of inputs, they give exactly the same output and send the circuit either to the same state or to an equivalent state. When two states are equivalent, one of them can be removed without altering the input-out relationships.*

The algorithm is illustrated using the state table of Table 10.4. Going through the state table, we look for two present states that go to the same next state and the same output for both input combinations. States *E* and *G* are two such states; they go to states *A* and *F* and have outputs of 0 and 1 for $x = 0$ and $x = 1$, respectively. Therefore, states *E* and *G* are equivalent; one can be removed. The procedure of removing a state and replacing it by its equivalent is demonstrated in Table 10.5.

The row with present state *G* is crossed out and state *G* is replaced by state *E* each time it occurs in the next state columns.

Table 10.4 An example state table.

Present state	Next state, Output z Input x 0	 1
A	*A*,0	*B*,0
B	*C*,0	*D*,0
C	*A*,0	*D*,0
D	*E*,0	*F*,1
E	*A*,0	*F*,1
F	*G*,0	*F*,1
G	*A*,0	*F*,1

Table 10.5 Reducing state table of Table 10.4.

Present state	Next state, Output z Input x 0	 1
A	*A*,0	*B*,0
B	*C*,0	*D*,0
C	*A*,0	*D*,0
D	*E*,0	*F*,1
E	*A*,0	*F*,1
F	*E* ~~*G*~~,0	*F*,1
~~*G*~~	*A*,0	*F*,1

Present state *F* now has next state *E* and *F* and outputs 0 and 1 for $x = 0$ and $x = 1$, respectively. The reduced state table is shown in Table 10.6.

Now, state *F* and *D* are equivalent and state *F* can be removed and replaced by *D*. The reduction of state is shown in Table 10.7. The final reduced table is shown in Table 10.8.

Table 10.6 Reduced state table for Table 10.4.

Present state	Next state, Output z Input x 0	 1
A	*A*,0	*B*,0
B	*C*,0	*D*,0
C	*A*,0	*D*,0
D	*E*,0	*F*,1
E	*A*,0	*F*,1
F	*E*,0	*F*,1

Table 10.7 Reducing state table of Table 10.6.

Present state	Next state, Output z Input x 0	 1
A	A,0	B,0
B	C,0	D,0
C	A,0	D,0
D	E,0	D ~~F~~,1
E	A,0	D ~~F~~,1
~~F~~	E,0	F,1

Table 10.8 Final reduced state table for Table 10.4.

Present state	Next state, Output z Input x 0	 1
A	A,0	B,0
B	C,0	D,0
C	A,0	D,0
D	E,0	D,1
E	A,0	D,1

It is worth noting that the reduction in the number of states of a sequential circuit is possible if one is interested only in external input-output relationships. When external outputs are taken directly from flip-flops, the output must be independent of the number of states before state reduction algorithm is applied.

The sequential circuit of this example was reduced from seven to five states. In either case, the representation of the states with physical components requires that we use three flip-flops, because m flip-flops can represent up to 2^m distinct states. With three flip-flops, we can formulate up to eight binary states denoted by binary numbers 000 through 111, with each bit designating the state of one flip-flop. If the state table of Table 10.4 is used, we must assign binary values to seven states; the remaining state is unused. If the state table of Table 10.8 is used, only five states need binary assignment, and we are left with three unused states. Unused states are treated as don't-care conditions during the design of the circuit. Since don't-care conditions usually help in obtaining a simpler Boolean function, it is more likely that the circuit with five states will require fewer combinational gates than the one with seven states. In any case, the reduction from seven to five states does not reduce the number of flip-flops. In general, reducing the number of states in a state table is likely to result in a circuit with less equipment. However, the fact that a state has been reduced to fewer states does not guarantee a saving in the number of flip-flops or the number of gates.

10.4.2 State Assignment

The cost of the combinational circuit part of a sequential circuit can be reduced by using the known simplification methods for combinational circuits. However, there is another factor, known as the *state assignment* problem, which comes into play in minimizing the combinational gates. State assignment procedures are concerned with methods for assigning binary values to states in such a way as to reduce the cost of the combinational circuit that drives the flip-flops. This is particularly helpful when a sequential circuit is viewed from its external input-output terminals. Such a circuit may follow a sequence of internal states, but the binary values of the individual states may be of no consequence as long as the circuit produces the required sequence of outputs for any given sequence of inputs. This does not apply to circuits whose external outputs are taken directly from flip-flops with binary sequences fully specified.

The binary state assignment alternatives available can be demonstrated in conjunction with the sequential circuit specified in Table 10.8. Remember that, in this example, the binary values of the states are immaterial as long as their sequence maintains the proper input-output relationships. For this reason, any binary number assignment is satisfactory as long as each state is assigned a unique number. Three examples of possible binary assignments are shown in Table 10.9 for the five states of the reduced table. Assignment 1 is a straight binary assignment for the sequence of states from A through E. The other two assignments are chosen arbitrarily.

Table 10.10 is the reduced transition table with binary assignment 1 substituting for the letter symbols of the five states. It is obvious that a different binary assignment will result in a transition table with different binary values for the states, while the input-output relationships remain the same. The transition table is used to derive the combinational circuit part of the sequential circuit. The complexity of the combninational circuit obtained depends on the binary state assignment chosen.

Table 10.9 Three binary state assignments to five states.

State	Assignment 1	Assignment 2	Assignment 2
A	001	000	000
B	010	010	100
C	011	011	010
D	100	101	101
E	101	111	011

Table 10.10 Reduced transition table of Table 10.8 with binary assignment 1 of Table 10.9.

Present state	Next state Input x 0	 1	Output z Input x 0	 1
001	001	010	0	0
010	011	100	0	0
011	001	100	0	0
100	101	100	0	1
101	001	100	0	1

Various procedures have been suggested that lead to a particular binary assignment from the many available. The most common criterion is that the chosen assignment should result in a simple combinational circuit for the flip-flop inputs. However, there are no state assignment procedures that guarantee a minimum-cost combinational circuit. State assignment is one of the challenging problems of digital logic design. Techniques for dealing with the state assignment problem are beyond the scope of this book.

10.5 Design Procedure of Sequential Circuit

The design of a sequential circuit starts from a set of specifications and culminates in a logic diagram. A sequential circuit can be designed using the following procedure:

1. The word description of the circuit behavior is stated. This may be accompanied by a state or transition diagram, a timing diagram, or other pertinent information.
2. From the given information about the circuit, obtain the state diagram (if not given in step 1) and state table.
3. Reduce, if possible, the number of states by state reduction method if the sequential circuit can be characterized by input-output relationships independent of the number of states.
4. Determine the number of flip-flops needed. Assign binary values to each state and obtain transition table.
5. Choose the type of flip-flops and from the transition table, derive the flip-flop excitation and output tables.
6. Using K-map or any other simplification method, derive the flip-flop excitation equations and output equations.
7. Draw the logic diagram.

The word specification of the circuit behavior usually uses many digital logic terminologies. It is necessary that the designer use intuition and experience to arrive at the correct interpretation of the circuit specifications, because word descriptions may be incomplete and inexact. However, once such a specification has been set down and the state table obtained, it is possible to make use of the formal procedure to design the circuit.

The design procedure is illustrated using the following examples.

Example 1:

Step 1. The sequential circuit to be designed is described by the transition diagram of Figure 10.6.

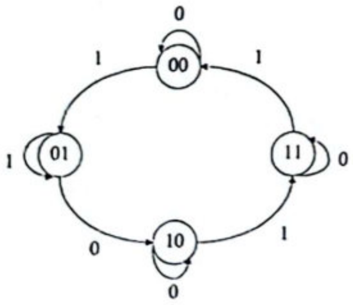


Figure 10.6 Transition diagram of a sequential circuit to be designed.

Step 2. The sequential circuit is described using a transition diagram. So, no state diagram and state table is needed.

Step 3. Since the directed lines in Figure 10.6 are marked with a single binary digit without a /, we conclude that there is one input variable and no output variables. The state of the flip-flops may be considered as the outputs of the circuit. Therefore, state reduction is not possible.

Step 4. The transition diagram of Figure 10.6 consists of four states with binary values already assigned. The two flip-flops needed are designated *FF*1 and *FF*2. The input variable is designated x. The transition table for this circuit is shown in Table 10.11. Note that there is no output section for this circuit.

Table 10.11 Transition table for the sequential circuit described by the transition diagram of Figure 10.6.

Present state $Q1_n Q2_n$	Next state $Q1_{n+1} Q2_{n+1}$ Input x	
	0	1
00	00	01
01	10	01
11	11	00
10	10	11

Step 5. The type of flip-flops to be used is J-K. The flip-flop excitation table is formed from the transition table of Table 10.11 and is shown in Table 10.12.

Table 10.12 Excitation table for the sequential circuit described by transition diagram of Figure 10.6.

Present state $Q1_n Q2_n$	Next state $Q1_{n+1} Q2_{n+1}$		Flip-flop inputs			
			$x=0$		$x=1$	
	$x=0$	$x=1$	$J1_n K1_n$	$J2_n K2_n$	$J1_n K1_n$	$J2_n K2_n$
00	00	01	0x	0x	0x	1x
01	10	01	1x	x1	0x	x0
11	11	00	x0	x0	x1	x1
10	10	11	x0	0x	x0	1x

The excitation table of a sequential circuit is a list of flip-flop input conditions that will cause the required state transitions and is a function of the type of flip-flop used. Since this example specified J-K flip-flops, we need columns for the *J* and *K* inputs of two flip-flops denoted by $J1_n K1_n$ and $J2_n K2_n$. The excitation table of the J-K flip-flop is given in Table 9.5. This table is used to derive the excitation table of the circuit. For example, in the first row of Table 10.12 we have a transition for flip-flop *FF*1 from 0 in the present state to 0 in the next state for $x = 0$. In Table 9.5 we find that a transition of state from 0 to 0 requires that input $J_n = 0$

and input $K_n = x$. So 0x is entered in the first row under $J1_n K1_n$ for $x = 0$. In a similar manner all the entries for flip-flop inputs are determined.

Step 6. The inputs to the combinational circuit are the external input (x) and the present state values ($Q1_n$ and $Q2_n$) of the flip-flops. The outputs of the combinational circuit go to flip-flop inputs ($J1_n$, $K1_n$, $J2_n$, and $K2_n$). The simplified Boolean functions for the combinational circuit can now be derived. The inputs are the variables $Q1_n$, $Q2_n$, and x; the outputs are the variables $J1_n$, $K1_n$, $J2_n$, and $K2_n$. The information from the excitation table is transferred into the K-maps of Figure 10.7, which give the simplified flip-flop excitation equations as below:

$$J1_n = Q2_n x'$$
$$K1_n = Q2_n x$$
$$J2_n = x$$
$$K2_n = Q1_n' x' + Q1_n x$$

As the circuit has no output, no output equation is derived.

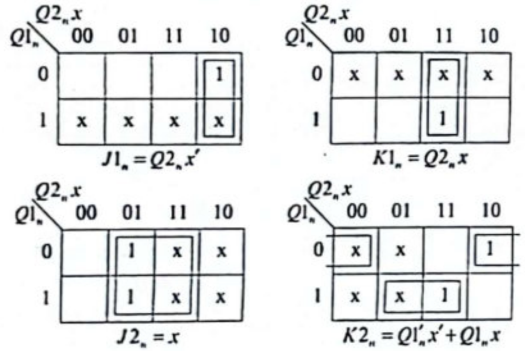


Figure 10.7 K-maps for the excitation equations for the sequential circuit specified by the transition diagram of Figure 10.6.

Step 7. The logic diagram is drawn in Figure 10.8.

Example 2:

Step 1. The sequential circuit is specified by the state table of Table 10.4.
Step 2. The state table is given as the description of the sequential circuit.
Step 3. The reduced state table is shown in Table 10.8.
Step 4. Three flip-flops are needed. The transition table is given in Table 10.10.
Step 5. The circuit is to employ S-R flip-flops. The excitation table of S-R flip-flop is given in Table 9.2. The excitation table of the sequential circuit is derived from the transition table of Table 10.10 in conjunction with Table 9.2 and is shown in Table 10.13.

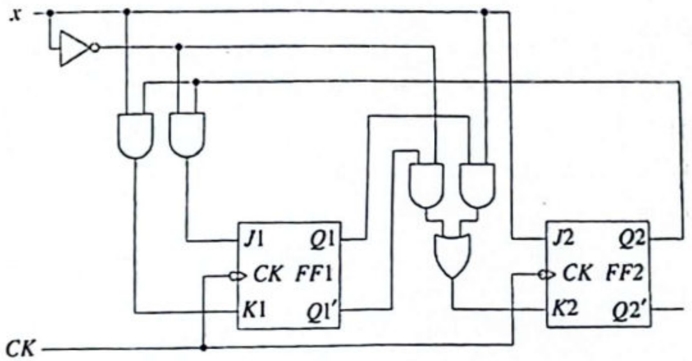


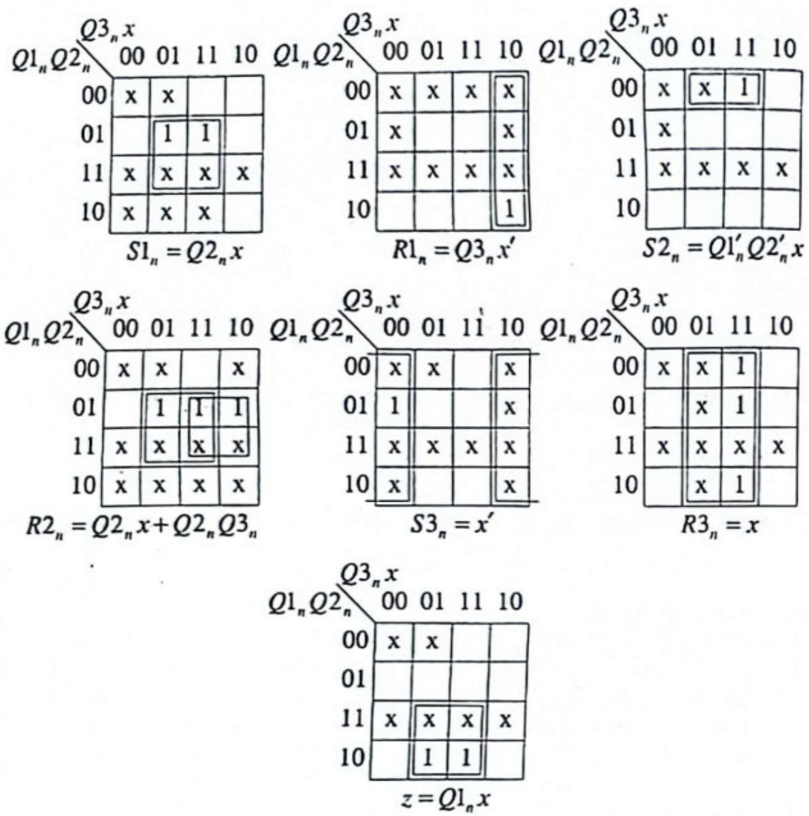
Figure 10.8 Logic diagram of sequential circuit specified by the transition diagram of Figure 10.6.

Table 10.13 Excitation for the sequential circuit specified by the state table of Table 10.4.

Present state $Q1_nQ2_nQ3_n$	Next state $Q1_{n+1}Q2_{n+1}Q3_{n+1}$		Flip-flop inputs						Output z	
			$x=0$			$x=1$			Input x	
	$x=0$	$x=1$	$S1_nR1_n$	$S2_nR2_n$	$S3_nR3_n$	$S1_nR1_n$	$S2_nR2_n$	$S3_nR3_n$	0	1
001	001	010	0x	0x	x0	0x	10	01	0	0
010	011	100	0x	x0	10	10	01	0x	0	0
011	001	100	0x	01	x0	10	01	01	0	0
100	101	100	x0	0x	10	x0	0x	0x	0	1
101	001	100	01	0x	x0	x0	0x	01	0	1

Step 6. The simplified flip-flop excitation equations and output equation are derived using the K-maps of Figure 10.9, which give

$$S1_n = Q2_n x$$
$$R1_n = Q3_n x'$$
$$S2_n = Q1_n' Q2_n' x$$
$$R2_n = Q2_n x + Q2_n Q3_n$$
$$S3_n = x'$$
$$R3_n = x$$
$$z = Q1_n x$$



$Q1_nQ2_n \backslash Q3_nx$	00	01	11	10
00	x	x		
01		1	1	
11	x	x	x	x
10	x	x	x	

$S1_n = Q2_nx$

$Q1_nQ2_n \backslash Q3_nx$	00	01	11	10
00	x	x	x	x
01	x			x
11	x	x	x	x
10				1

$R1_n = Q3_nx'$

$Q1_nQ2_n \backslash Q3_nx$	00	01	11	10
00	x	x	1	
01	x			
11	x	x	x	x
10				

$S2_n = Q1'_nQ2'_nx$

$Q1_nQ2_n \backslash Q3_nx$	00	01	11	10
00	x	x		x
01		1	1	1
11	x	x	x	x
10	x	x	x	x

$R2_n = Q2_nx + Q2_nQ3_n$

$Q1_nQ2_n \backslash Q3_nx$	00	01	11	10
00	x	x		x
01	1			x
11	x	x	x	x
10	x			x

$S3_n = x'$

$Q1_nQ2_n \backslash Q3_nx$	00	01	11	10
00	x	x	1	
01		x	1	
11	x	x	x	x
10		x	1	

$R3_n = x$

$Q1_nQ2_n \backslash Q3_nx$	00	01	11	10
00	x	x		
01				
11	x	x	x	x
10		1	1	

$z = Q1_nx$

Figure 10.9 K-maps for the excitation equations and output equation of the sequential circuit specified by the state table of Table 10.4.

There are three unused states in this circuit: binary states 000, 110, and 111. When an input of 0 or 1 is included with these unused states, we obtain six don't-care minterms: 0, 1, 12, 13, 14, and 15. These six binary combinations are not listed in Table 10.13 and are treated as don-t-care terms.

Step 7. The logic diagram is shown in Figure 10.10.

Example 3:

Step 1. A sequential circuit is to be designed, which has a single input and single output. The output is to be zero unless an input sequence 0010 is received. For example, if the input sequence is 0100100100, the output sequence is 0000010010.

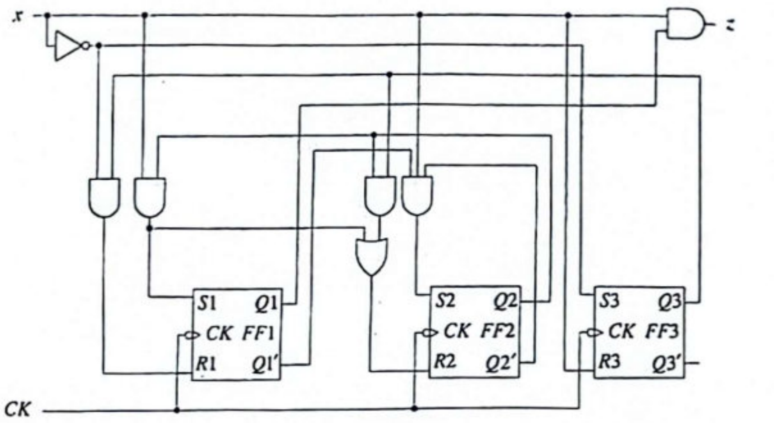


Figure 10.10 Logic diagram of the sequential circuit specified by the state table of . Table 10.4.

Step 2. The state diagram for this problem is shown in Figure 10.11. To make a state diagram, we must begin with an initial state designated as *A*. The initial state *A* is generally selected as a shortest sequence which can not be the part of a successful sequence (namely 0010). In this case 11 is such a shortest sequence and complete sequence (0010) must follow 11 to produce an output. Starting with *A*, the circuit can receive either a 1 or 0. There is no change in state if a 1 is received and a self-loop is drawn at *A* labeled as 1/0. If a 0 is applied, the circuit goes to a new state *B* with an output 0. When the circuit is in state *B*, a 1 input takes it to a new state *C* while a 0 input takes it to another state *D*. When the circuit is in the state *D*, its last two inputs are 00 and another 0 applied does not result in any new state and it is shown by a self-loop at *D*. The circuit goes to a new state *E* if a 1 is applied to it. When the circuit is in state *C* and a 1 is applied, the input sequence becomes 011 and the circuit therefore goes back to the state *A*. A 0 applied to the circuit however takes it to state *B* since only one 0 has been applied to the circuit as a part of the successful sequence. When the circuit is in the state *E*, a further 0 results in the completion of the sequence and the output is 1. However a 1 applied to the circuit takes it back to *A*. After the completion of successful sequence, the state *F* results and a 0 applied in this state takes it to state *D* since another 10 as input will result in the completion of sequence. A 1 applied however will take it to state *C* because the fresh sequence has to be started again.

The state table for this circuit is given in Table 10.14.

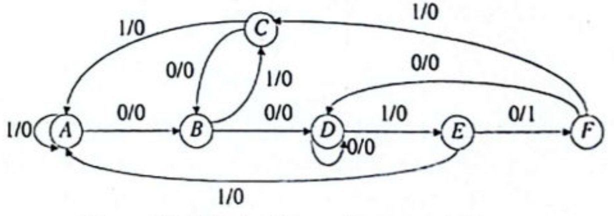


Figure 10.11 State diagram for example 3.

Table 10.14 State table for the sequential circuit specified by the state diagram of Figure 10.11.

Present state	Next state, output z Input x 0	 1
A	B,0	A,0
B	D,0	C,0
C	B,0	A,0
D	D,0	E,0
E	F,1	A,0
F	D,0	C,0

Step 3. In this problem the input-output relationship is independent of the number of states. Therefore, state reduction is possible. In table 10.14, present states A and C are equivalent and states B and F are equivalent. The state reduction is shown in Table 10.15.

Table 10.15 Reducing state table of Table 10.14.

Present state	Next state, output z Input x 0	 1
A	B,0	A,0
B	D,0	A ~~C~~,0
~~C~~	B,0	A,0
D	D,0	E,0
E	B ~~F~~,1	A,0
~~F~~	D,0	A ~~C~~,0

The reduced state table is given in Table 10.16.

Table 10.16 Reduced state table of Table 10.14.

Present state	Next state, output z Input x 0	 1
A	B,0	A,0
B	D,0	A,0
D	D,0	E,0
E	B,1	A,0

Step 4. The reduced state table has four states. Therefore, two flip-flops are needed. The states are assigned binary values arbitrarily as follows: $A = 00$, $B = 01$, $D = 11$, and $E = 10$. Using this binary assignment the transition table is shown in Table 10.17.

Table 10.17 Transition table corresponding to the state table of Table 10.16.

Present state $Q1_nQ2_n$	Next state $Q1_{n+1}Q2_{n+1}$ $x=0$	 $x=1$	Output z $x=0$	 $x=1$
00	01	00	0	0
01	11	00	0	0
11	11	10	0	0
10	01	00	1	0

Step 5. J-K flip-flops are chosen to be used for implementing the circuit. The flip-flop excitation table and output table are shown in Table 10.18.

Table 10.18 Excitation and output table for example 3.

Present state	Next state $Q1_{n+1}Q2_{n+1}$		Flip-flops inputs				Output z	
$Q1_nQ2_n$	$x=0$	$x=1$	$x=0$		$x=1$		$x=0$	$x=1$
			$J1_nK1_n$	$J2_nK2_n$	$J1_nK1_n$	$J2_nK2_n$		
00	01	00	0x	1x	0x	0x	0	0
01	11	00	1x	x0	0x	x1	0	0
11	11	10	x0	x0	x0	x1	0	0
10	01	00	x1	1x	x1	0x	1	0

Step 6. The simplified flip-flop excitation equations and output equation are derived using the K-maps of Figure 10.12. In the K-maps the unused states are treated as don't-care conditions. The equations are as below:

Step 7. The logic diagram is shown in Figure 10.13.

$$J1_n = Q2_n x'$$

$$K1_n = Q2'_n$$

$$J2_n = x'$$

$$K2_n = x$$

$$z = Q1_n Q2'_n x'$$

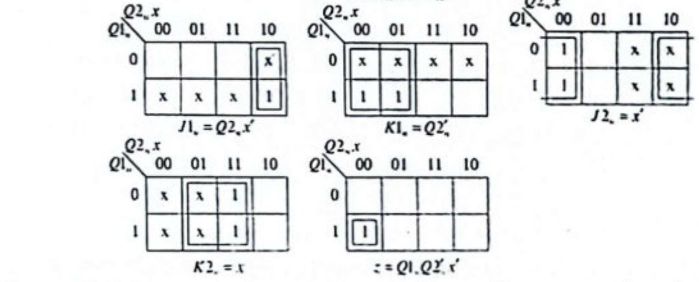


Figure 10.12 K-maps for the excitation equations and output equation of example 3.

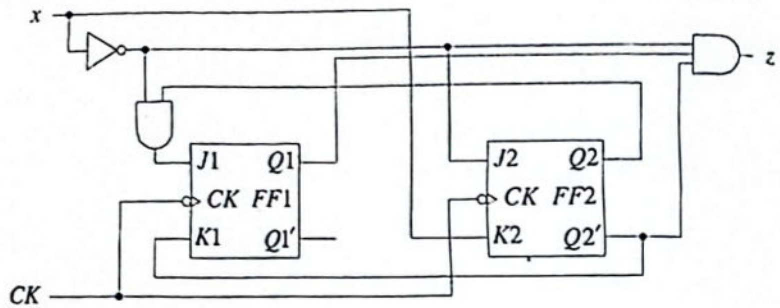


Figure 10.13 Logic diagram for example 3.

One factor neglected up to this point in the design is the initial state of a sequential circuit. When power is first turned on in a sequential circuit, one does not know in what state the flip-flops will settle. It is customary to provide a *master-reset* input whose purpose is to initialize the states of all flip-flops in the circuit. Typically, the master reset is a signal applied to all flip-flops asynchronously before the clocked operations start. In most cases flip-flops are cleared to 0 by the master-reset signal, but some may be set to 1. For example, the circuit of Figure 10.10 may initially be reset to a state $Q1Q2Q3 = 001$, since state 000 is not a valid state for this circuit. Figure 10.10 is redrawn in Figure 10.14 with master-reset input *RST*. A 0 in the *RST* input asynchronously resets the states of the circuit to 001. A 1 input in the *RST* input brings the circuit into normal operations.

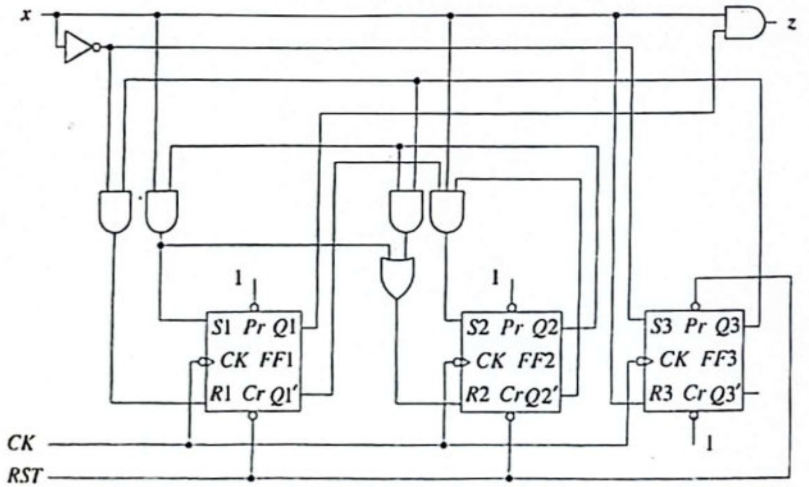


Figure 10.14 Logic diagram of Figure 10.10 with *RST* input to asynchronously reset the state to 001.

It was stated previously that unused states in a sequential circuit could be treated as don't-care conditions. Once the circuit is designed, the *m* flip-flops in the circuit can be in any one of the 2^m possible states. If some of these states were taken as don't-care conditions, the circuit must be investigated to determine the effect of these unused-states. The next state from invalid states can be determined from the analysis of the circuit. In any case, it is always wise to analyze a circuit obtained from a design to ensure that no mistakes were made during the design process.

In Example 2, unused states 000, 110, and 111 are treated as don't-care conditions. The effect of these unused states is determined by analyzing the sequential circuit of Figure 10.10. The excitation equations and output equation of the circuit (see Figure 10.9) are

$$\begin{aligned}
S1_n &= Q2_n x \\
R1_n &= Q3_n x' \\
S2_n &= Q1'_n Q2'_n x \\
R2_n &= Q2_n x + Q2_n Q3_n \\
S3_n &= x' \\
R3_n &= x \\
z &= Q1_n x
\end{aligned}$$

The next state equations of the flip-flops are obtained from the excitation equations as below:

$$\begin{aligned}
Q1_{n+1} &= Q2_n x + (Q3_n x')' Q1_n \\
&= Q2_n x + Q1_n Q3'_n + Q1_n x \\
Q2_{n+1} &= Q1'_n Q2'_n x + (Q2_n x + Q2_n Q3_n)' Q2_n \\
&= Q1'_n Q2'_n x + (Q2_n x)'(Q2_n Q3_n)' Q2_n \\
&= Q1'_n Q2'_n x + (Q2'_n + x')(Q2'_n + Q3'_n) Q2_n \\
&= Q1'_n Q2'_n x + Q2_n Q3'_n x' \\
Q3_{n+1} &= x' + x' Q3_n \\
&= x'
\end{aligned}$$

Transition table is prepared from the next state equations and the output equation and is given in Table 10.19.

Table 10.19 Transition table of the sequential circuit of Figure 10.10.

Present state $Q1_nQ2_nQ3_n$	Next state $Q1_{n+1}Q2_{n+1}Q3_{n+1}$ $x=0$	$x=1$	Output z $x=0$	$x=1$
000	001	010	0	0
001	001	010	0	0
010	011	100	0	0
011	001	100	0	0
100	101	100	0	1
101	001	100	0	1
110	111	100	0	1
111	001	100	0	1

The transition diagram is drawn from the transition table of Table 10.19 and is shown in Figure 10.15.

From the analysis we found that the second to sixth rows of Table 10.19 is exactly identical to the Table 10.10. Therefore, the design of Example 2 is correct. Now we will determine the effect of the unused states 000, 110, and 111. From the transition diagram of Figure 10.15, we found that the circuit operates as intended as long as it stays within the state 001, 010, 011, 100, and 101. If it ever finds itself in one of the invalid states 000, 110, and 111, it goes to one of the valid states within one or two clock pulses. Thus the circuit is *self-starting* and *self-correcting*, since it eventually goes to a valid state from which it continues to operate as required.

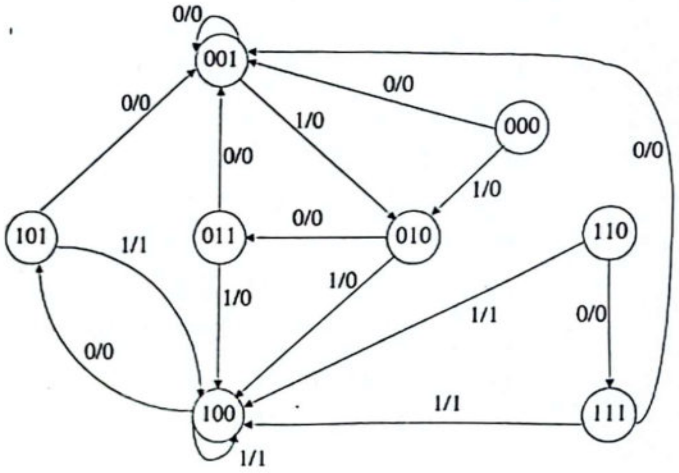

0/0
001
0/0
0/0
1/0
000
0/0
0/0
1/0
101
1/1
011
0/0
010
110
1/0
1/0
1/1
0/0
0/0
100
1/1
111
1/1

Figure 10.15 Transition diagram for the sequential circuit of Figure 13.10.

Exercises

10.1 For the sequential circuit of Figure 10.16 obtain the (a) state equations, (b) transition table, (c) state table, and (d) state diagram.

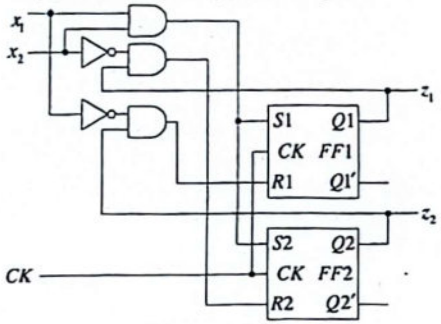

x_1
x_2
z_1
S1 Q1
CK FF1
R1 Q1'
z_2
S2 Q2
CK
CK FF2
R2 Q2'

Figure 10.16 Logic diagram for Exercise 10.1.

10.2 Analyze the sequential circuit of Figure 10.17 by drawing transition table, state table, and state diagram.

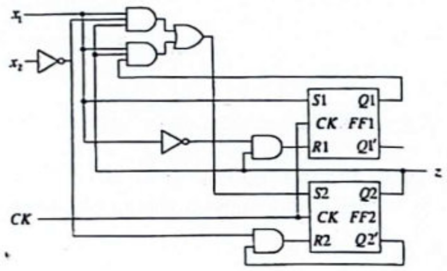

x_1
x_2
S1 Q1
CK FF1
R1 Q1'
z
S2 Q2
CK
CK FF2
R2 Q2'

Figure 10.17 Logic diagram for Exercise 10.2.

10.3 Analyze the sequential circuit of Figure 10.18 by forming transition table, state table, and state diagram.

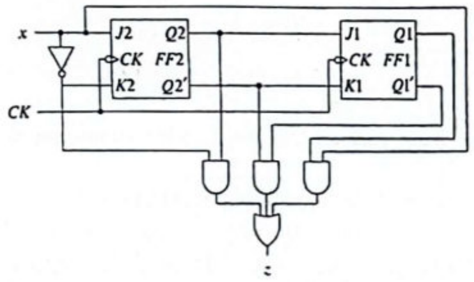

x
J2 Q2
CK FF2
K2 Q2'
J1 Q1
CK FF1
K1 Q1'
CK
z

Figure 10.18 Logic diagram for Exercise 10.3.

10.4 Analyze the sequential circuit described by the following next state and output equations:

$$Q1_{n+1} = x_1x_2 + x_2Q1_n + Q1_nQ2'_n$$
$$Q2_{n+1} = x_1Q1_n + x_2Q1'_n + Q1'_nQ2_n$$
$$z = x_1Q1'_nQ2_n + x'_2Q1_nQ2_n + x'_1x_2Q1_n$$

10.5 The full-adder of Figure 10.19 receives two external inputs *A* and *B*; the third input is *Z* comes from the output of a D flip-flop. The carry output is transferred to the flip-flop at every clock pulse. The external output *S* gives the sum of *A*, *B*, and *Z*. Obtain the transition table and transition diagram of the sequential circuit.

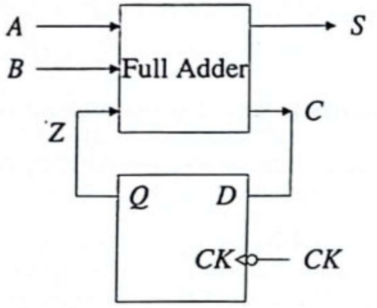


Figure 10.19 Logic diagram for Exercise 10.5.

10.6 A sequential circuit has two flip-flops (*FF*1 and *FF*2), two inputs (*x* and *y*), and an output (*z*). The flip-flop input equations and the output equation are as follows:

$$J1_n = xQ2_n + y'Q2'_n$$
$$K1_n = xy'Q2'_n$$
$$J2_n = xQ1'_n$$
$$K2_n = xy' + Q1_n$$
$$z = xyQ1_n + x'y'Q2_n$$

Obtain the logic diagram, transition table, transition diagram, and state equations.

10.7 Design a sequential circuit with one input line *x* and a single output line *z*. The output is obtained ($z = 1$) whenever the sequence 0110 is received. For example, if the input sequence is 000110110 the output sequence would be 000001001. Use S-R flip-flops.

10.8 Repeat Exercise 10.7 using J-K flip-flops.

10.9 Find out a minimal state table equivalent to the state table of Table 10.20.

Table 10.20 State table for Exercise 10.9.

Present state	Next state, Output Input	
	0	1
A	*G*,0	*B*,0
B	*F*,1	*I*,1
C	*H*,1	*E*,0
D	*F*,1	*C*,1
E	*G*,1	*C*,0
F	*D*,1	*E*,1
G	*A*,0	*F*,0
H	*G*,0	*B*,0
I	*A*,1	*E*,0

10.10 Determine a minimal state table equivalent to the state table of Table 10.21.

Table 10.21 State table for Exercise 10.10.

Present state	Next state, Output Input	
	0	1
A	*A*,0	*A*,0
B	*A*,1	*F*,1
C	*D*,0	*E*,0
D	*A*,1	*G*,0
E	*B*,0	*C*,0
F	*D*,0	*E*,0
G	*B*,0	*C*,0

10.11 A sequential circuit has one input and one output. The transition diagram is shown in Figure 10.20. Design the sequential circuit with T flip-flops.

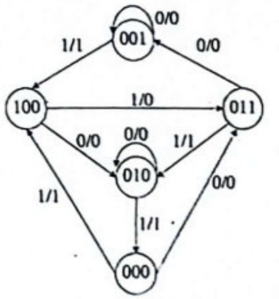


Figure 10.20 Transition diagram for Exercise 10.11.

Further Reading

1. E. J. McCluskey, *Logic Design Proncirples*, Prentice-Hall, Inc., 1986.
2. Zvi Kohavi, *Switching and Finite Automata Theory*, 2nd ed., McGraw-Hill Higher Education, 1990.
3. Richard S. Sandige, *Modern Digital Design*, McGraw-Hill Publishing Company, 1990.
4. R. P. Jain, *Modern Digital Electronics*, 2nd ed., Tata McGraw-Hill Publishing Company Limited, India, 1998.
5. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
6. John P. Uyemura, *Digital Systems Design: An Integrated Approach*, Brooks/Cole Publishing Company, 2000.
7. V. K. Jain, *Switching Theory and Digital Electronics*, Khanna Publishers, India, 2002.
8. M. Morris Mano, *Digital Design*, 3rd ed., Pearson Education, 2002.
9. Samuel C. Lee, *Digital Circuits and Logic Design*, Prentice-Hall, Inc., 2002.

CHAPTER 11
REGISTERS AND COUNTERS

11.1 Introduction

A *register* is a group of binary storage cells suitable for holding binary information. A group of flip-flops constitute a register, since each flip-flop is a binary cell capable of storing one bit of information. An *n*-bit register has a group of *n* flip-flops and is capable of storing any binary information containing *n* bits. In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks. Registers are extensively used in the design of digital systems in general and digital computers in particular.

A sequential circuit that goes through a prescribed sequence of states upon the application of input pulses is called a *counter*. The input pulses, called *count pulses*, may be clock pulses, or they may originate from an external source and may occur at prescribed intervals of time or at random. A counter is essentially a register that goes through a predetermined sequence of states upon the application of input pulses. The gates in a counter are connected in such a way as to produce a prescribed sequence of binary states in the register. Although counters are a special type of registers, it is common to differentiate them by giving them a special name. Counters are extensively used in the design of digital systems in general and digital computers in particular. Counters are used for counting the number of occurrences of an event and useful for generating timing variables to sequence and control the operations in a digital system.

In this chapter the design of registers is discussed along with a list of available MSI register ICs. Some of the common applications of registers are also discussed. The design of counters is also discussed followed by brief introduction to MSI counter ICs. Applications of counters are also discussed.

11.2 Registers and Their Classification

A *register* is a group of binary storage cells suitable for holding binary information. An *n*-bit register has a group of *n* flip-flops and is capable of storing any binary information containing *n* bits. In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks. In registers, the data can be entered in *serial* (one-bit at a time) or in *parallel* (all the bits simultaneously) form and can be retrieved in the serial or parallel form. A 4-bit data 1010 in serial form is shown in Figure 11.1(a) and in parallel form in Figure 11.1(b). For serial input/output, only one line is required for data input and one line for data output, whereas the number of lines required is equal to the number of bits for parallel input/output.

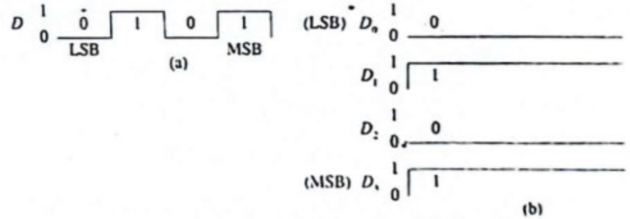


Figure 11.1 Data representation in (a) serial form and (b) parallel form.

Registers are classified depending upon the way in which data are entered and retrieved. There are four possible modes of operation:

1. Serial-in, serial-out (SISO)
2. Serial-in, parallel-out (SIPO)
3. Parallel-in, parallel-out (PIPO)
4. Parallel-in, serial-out (PISO)

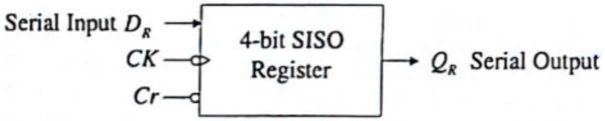
Registers in which data are entered or/and taken out in serial form are referred to as *shift registers*, since bits are shifted in the flip-flops with the occurrence of clock pulses. Data shifts may be either in the right direction (*right-shift register*) or in the left direction (*left-shift register*). Data can also be shifted from left to right as well as in the reverse direction (*bi-directional shift register*) using the mode control input.

A register is referred to as a *universal register* if it can be operated in all the four possible modes and also as a bi-directional register.

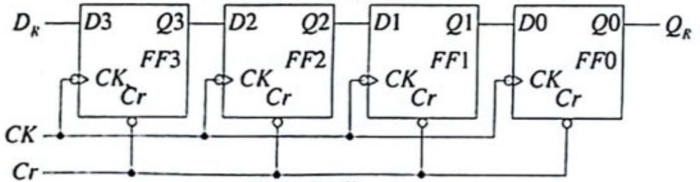
11.3 Serial-In Serial-Out Registers

11.3.1 Serial-In Serial-Out Right-Shift Register

The block diagram and a logic diagram of a 4-bit serial-in serial-out right-shift register are shown in Figure 11.2. The Q output of a given flip-flop is connected to the D input of the flip-flop at its right. The serial input (D_R) determines what goes into the left most flip-flop during the shift. The serial output (Q_R) is taken from the output of the right most flip-flop prior to the application of a clock pulse. The data word in the serial form (Figure 11.1(a)) is applied at the serial input (D_R) after clearing the flip-flops using the clear (*Cr*) input. The input and output waveform are illustrated in Figure 11.3 for a data word 0110.



(a) Block diagram



(b) Logic diagram

Figure 11.2 A 4-bit serial-in serial-out right-shift register.

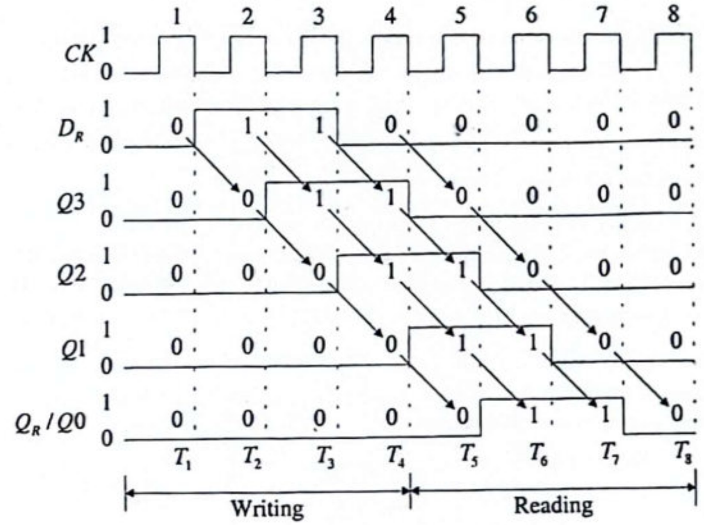


Figure 11.3 Waveforms of serial-in serial-out right-shift register.

The process of entering (writing) the data word starts with the data input corresponding to the least significant bit (0 in this case) at the serial input and first clock pulse. At the falling edge (T_1) of the first clock pulse the output of *FF*3 will be 0 and the outputs of the other flip-flops are 0 since their inputs are 0. Next, the input corresponding to the next bit (1 in this case) is applied and, at the falling edge (T_2) of the second clock pulse, the flip-flop outputs will be

$$Q3 = 1$$

$$Q2 = Q1 = Q0 = 0$$

Similarly, the input corresponding to each bit is applied till the MSB and the bits go on shifting from left to right at the falling edge of each clock pulse as illustrated in Figure 11.3. At the end of the fourth clock pulse (T_4), the outputs of flip-flops are

$$Q3 = 0$$

$$Q2 = 1$$

$$Q1 = 1$$

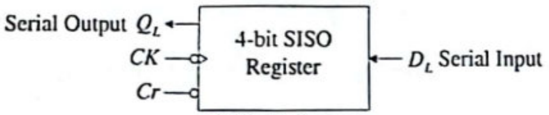
$$Q0 = 0$$

which is the same as the number to be stored. The number of clock pulses required for entering the data is the same as the number of bits.

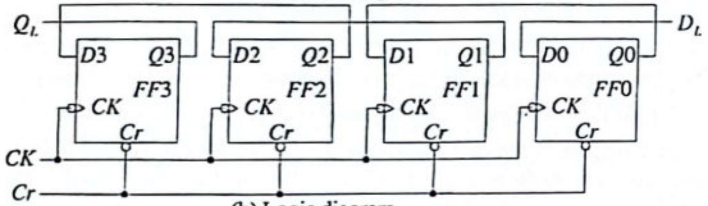
The data in the serial form can be retrieved (read) at $Q0$ (Q_R output) when clock pulses are applied. The number of clock pulses required will be same as the number of bits (four in this case). After each clock pulse flip-flop output will be changed depending on the value at the D_R input. This means that data can be retrieved from a SISO register only once.

11.3.2 Serial-In Serial-Out Left-Shift Register

Block diagram and a logic diagram of a 4-bit serial-in serial-out left-shift register are shown in Figure 11.4. The Q output of a given flip-flop is connected to the D input of the flip-flop at its left. The serial input (D_L) is at the D input of the right most flip-flop and the serial out (Q_L) is at the Q output of the left most flip-flop. The operation is similar to that of serial-in serial-out right-shift register, except that the data shift occurs in the left direction.


Serial Output Q_L
CK
Cr
4-bit SISO Register
D_L Serial Input

(a) Block diagram


Q_L
D_L
D3 Q3 FF3 CK Cr
D2 Q2 FF2 CK Cr
D1 Q1 FF1 CK Cr
D0 Q0 FF0 CK Cr
CK
Cr

(b) Logic diagram

Figure 11.4 A 4-bit serial-in serial-out left-shift register.

11.3.3 Serial-In Serial-Out Bi-Directional Shift Register

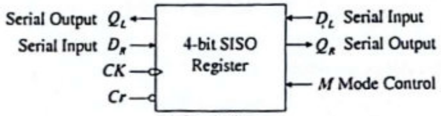
There are applications in which shifting data to the right and/or to the left is required. For example, a binary number can be divided by two by shifting it one stage to the right. In this process the least significant bit is lost (unless additional circuitry is used to preserve it) causing an error of 0.5 if the number is odd. Similarly, a number stored in a shift register can be multiplied by two by shifting it one stage to the left, provided a 1 is not shifted out of the most significant stage.

The block diagram and a logic diagram of a 4-bit serial-in serial-out bi-directional shift register are shown in Figure 11.5. When the mode control $M = 1$, all the A AND gates are enabled and the data D_R is shifted to the right when clock pulses are applied. On the other hand, when $M = 0$, the A gates are inhibited and B gates are enabled allowing the data at D_L to be shifted to the left. M should be changed only when $CK = 0$, otherwise the data stored in the register may be altered.

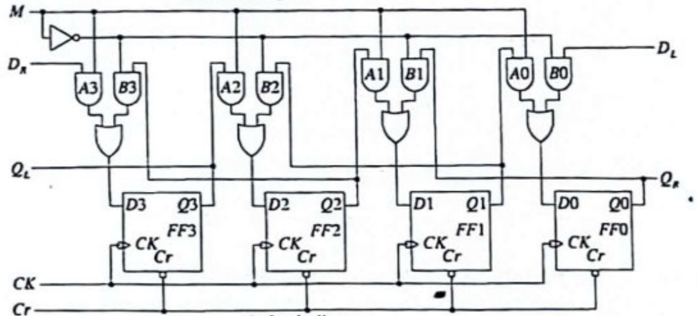
11.4 Serial-In Parallel-Out Registers

11.4.1 Serial-In Parallel-Out Right-Shift Register

The block diagram and a logic diagram of serial-in parallel-out right-shift register are shown in Figure 11.6. The data is entered into the register as a serial data through the serial input (D_R) as in Figure 11.3. After entering the data, it is available at the parallel outputs ($Q_3Q_2Q_1Q_0$) and no clock pulse is required for reading the data. The content of the register can be read any number of times until new data is entered into the register.

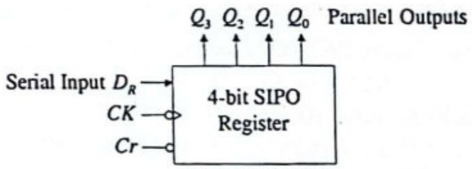

Serial Output Q_L
Serial Input D_R
CK
Cr
4-bit SISO Register
D_L Serial Input
Q_R Serial Output
M Mode Control

(a) Block diagram

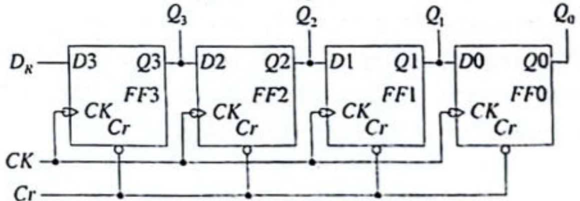

M
D_R
D_L
A3 B3
A2 B2
A1 B1
A0 B0
Q_L
Q_R
D3 Q3 FF3 CK Cr
D2 Q2 FF2 CK Cr
D1 Q1 FF1 CK Cr
D0 Q0 FF0 CK Cr
CK
Cr

(b) Logic diagram

Figure 11.5 A 4-bit serial-in serial-out bi-directional shift register.


Q_3 Q_2 Q_1 Q_0 Parallel Outputs
Serial Input D_R
CK
Cr
4-bit SIPO Register

(a) Block diagram

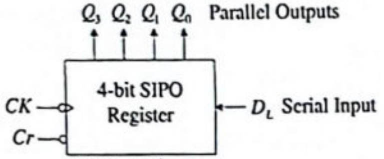

Q_3
Q_2
Q_1
Q_0
D_R
D3 Q3 FF3 CK Cr
D2 Q2 FF2 CK Cr
D1 Q1 FF1 CK Cr
D0 Q0 FF0 CK Cr
CK
Cr

(b) Logic diagram

Figure 11.6 A 4-bit serial-in parallel-out right-shift register.

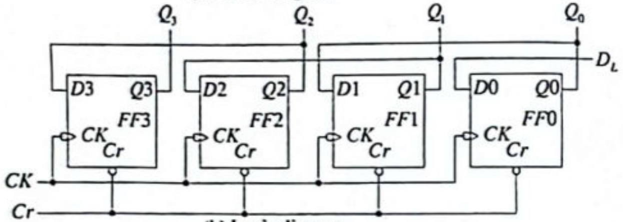
11.4.2 Serial-In Parallel-Out Left-Shift Register

The block diagram and a logic diagram of a 4-bit serial-in parallel-out left-shift register are shown in Figure 11.7. The circuit is similar to the circuit of serial-in serial-out left-shift register of Figure 11.4, except that the output is taken from $Q_3Q_2Q_1Q_0$ outputs in parallel. The operation is similar to that of Figure 11.6.



Q_3 Q_2 Q_1 Q_0 Parallel Outputs
4-bit SIPO Register
CK
Cr
D_L Serial Input

(a) Block diagram



Q_3
Q_2
Q_1
Q_0
D_L
D3 Q3 FF3 CK Cr
D2 Q2 FF2 CK Cr
D1 Q1 FF1 CK Cr
D0 Q0 FF0 CK Cr
CK
Cr

(b) Logic diagram

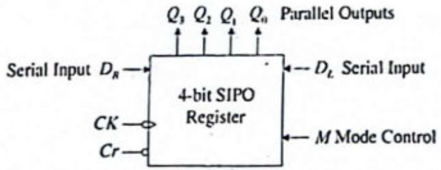
Figure 11.7 A 4-bit serial-in parallel-out left-shift register.

11.4.3 Serial-In Parallel-Out Bi-Directional Shift Register

The block diagram and a logic diagram of 4-bit serial-in parallel-out bi-direction shift register are shown in Figure 11.8. The circuit is similar to the circuit of serial-in serial-out bi-directional shift register of Figure 11.5, except that the output is taken from $Q_3Q_2Q_1Q_0$ outputs in parallel. The data is entered as serial data as usual. The direction of data shift is controlled by the Mode Control (M) input.

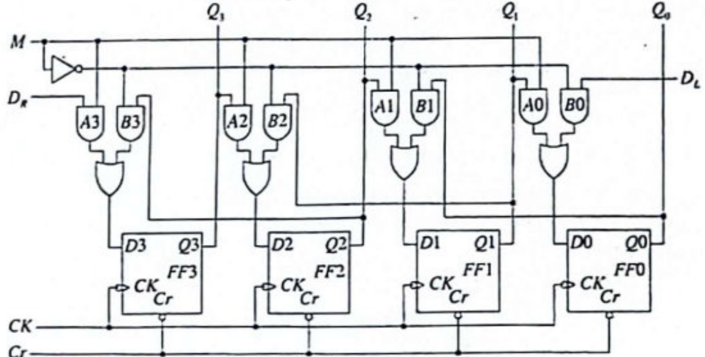
11.5 Parallel-In Parallel-Out Register

The block diagram and a logic diagram of a 4-bit parallel-in parallel-out register are shown in Figure 11.9. The register is constructed using D flip-flops. The data word in the parallel form (Figure 11.1(b)) is applied at the parallel inputs ($D_3D_2D_1D_0$). When a clock pulse is applied the data is entered into the register. This type of entering data is referred to as *synchronous loading*. After synchronous loading the data is available at the parallel outputs ($Q_3Q_2Q_1Q_0$).



Q_3 Q_2 Q_1 Q_0 Parallel Outputs
Serial Input D_R
D_L Serial Input
4-bit SIPO Register
CK
Cr
M Mode Control

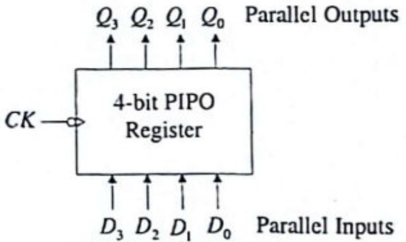
(a) Block diagram



Q_3
Q_2
Q_1
Q_0
M
D_R
D_L
A3 B3
A2 B2
A1 B1
A0 B0
D3 Q3 FF3 CK Cr
D2 Q2 FF2 CK Cr
D1 Q1 FF1 CK Cr
D0 Q0 FF0 CK Cr
CK
Cr

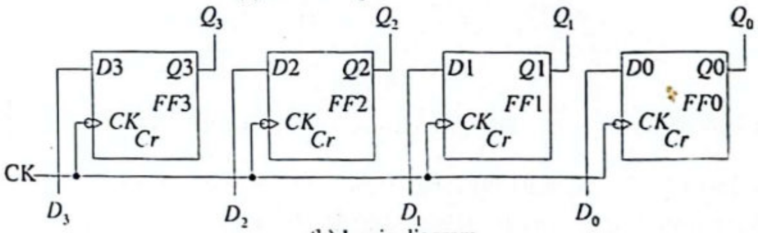
(b) Logic diagram

Figure 11.8 A 4-bit serial-in parallel-out bi-directional shift register.



Q_3 Q_2 Q_1 Q_0 Parallel Outputs
4-bit PIPO Register
CK
D_3 D_2 D_1 D_0 Parallel Inputs

(a) Block diagram

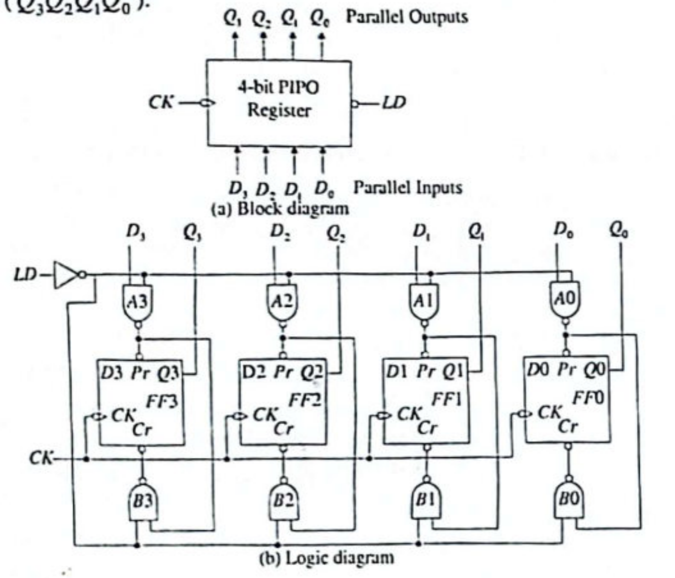


Q_3
Q_2
Q_1
Q_0
D3 Q3 FF3 CK Cr
D2 Q2 FF2 CK Cr
D1 Q1 FF1 CK Cr
D0 Q0 FF0 CK Cr
CK
D_3
D_2
D_1
D_0

(b) Logic diagram

Figure 11.9 A 4-bit parallel-in parallel-out register.

Another block diagram and a logic diagram for a 4-bit parallel-in parallel-out register are shown in Figure 11.10. The register is constructed using D flip-flops. In this circuit any type of flip-flops can be used, since the parallel input is performed using asynchronous *Pr* and *Cr* inputs of the flip-flops. In this case the data entering is *asynchronous*, that is, data is entered when the clock is 0. The data word in the parallel form is applied at the parallel inputs ($D_3D_2D_1D_0$). When the load input (*LD*) is 0, the output of the *A* NAND gates are complement of the data word and the output of the *B* NAND gates are equal to the data word. Consequently, the data word is entered into the register. For example, if $D_0 = 0$, then for flip-flop *FF*0, $\text{Pr} = 1$ and $Cr = 0$ and a 0 is entered into the Flip-flop *FF*0, that is $Q_0 = 0$. On the other hand, if $D_0 = 1$, then for flip-flop *FF*0, $\text{Pr} = 0$ and $Cr = 1$ and a 1 is entered into the flip-flop *FF*0. This type of entering data is referred to as *asynchronous loading*. After asynchronous loading, the data word is available at the parallel outputs ($Q_3Q_2Q_1Q_0$).



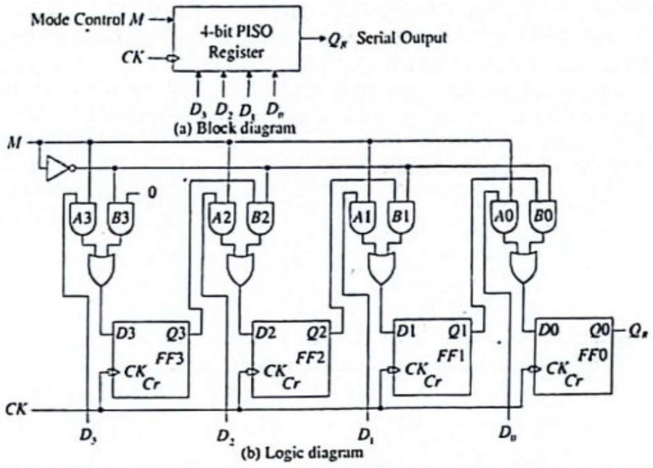
Q3 Q2 Q1 Q0 Parallel Outputs
CK
4-bit PIPO Register
LD
D3 D2 D1 D0 Parallel Inputs
(a) Block diagram
D3 Q3 D2 Q2 D1 Q1 D0 Q0
LD
A3 A2 A1 A0
D3 Pr Q3 FF3 CK Cr
D2 Pr Q2 FF2 CK Cr
D1 Pr Q1 FF1 CK Cr
D0 Pr Q0 FF0 CK Cr
CK
B3 B2 B1 B0
(b) Logic diagram

Figure 11.10 A 4-bit parallel-in parallel-out register (asynchronous loading).

11.6 Parallel-In Serial-Out Registers

11.6.1 Parallel-In Serial-Out Right-Shift Register

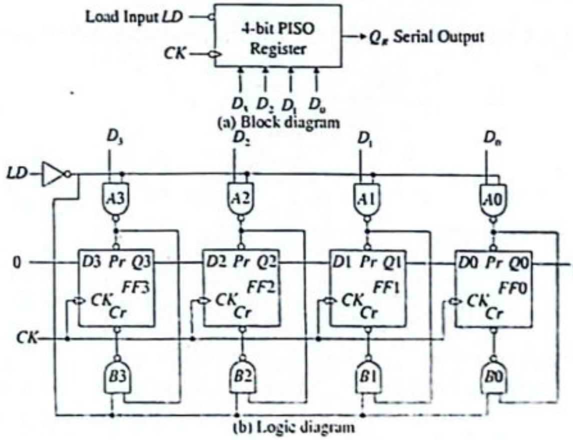
The block diagram and a logic diagram for a 4-bit parallel-in serial-out right-shift register with synchronous loading are shown in Figure 11.11. The register is constructed with D flip-flops. When the mode select input (*M*) is 1, all *A* AND gates are enabled and data at the parallel inputs ($D_3D_2D_1D_0$) is loaded into the flip-flops after clock pulse is applied. When *M* is 0, all *B* AND gates are enabled and the register becomes a right-shift register.



Mode Control M
CK
4-bit PISO Register
Q_R Serial Output
D3 D2 D1 D0
(a) Block diagram
M
0
A3 B3 A2 B2 A1 B1 A0 B0
D3 Q3 FF3 CK Cr
D2 Q2 FF2 CK Cr
D1 Q1 FF1 CK Cr
D0 Q0 FF0 CK Cr
Q_R
CK
D3 D2 D1 D0
(b) Logic diagram

Figure 11.11 A 4-bit parallel-in serial-out right-shift register (synchronous loading).

Another block diagram and a logic diagram for a 4-bit parallel-in serial-out right-shift register with asynchronous loading are shown in Figure 11.12. The register is constructed with D flip-flops. When the load input (*LD*) is 0 the data at the parallel-inputs ($D_3D_2D_1D_0$) is asynchronously loaded into the flip-flops, that is, the data is loaded when the clock is 0. When the *LD* input is 1 the register becomes serial-out right-shift register.



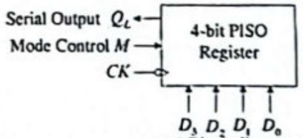
Load Input LD
CK
4-bit PISO Register
Q_R Serial Output
D3 D2 D1 D0
(a) Block diagram
D3 D2 D1 D0
LD
A3 A2 A1 A0
0
D3 Pr Q3 FF3 CK Cr
D2 Pr Q2 FF2 CK Cr
D1 Pr Q1 FF1 CK Cr
D0 Pr Q0 FF0 CK Cr
Q_R
CK
B3 B2 B1 B0
(b) Logic diagram

Figure 11.12 A 4-bit parallel-in serial-out right-shift register (asynchronous loading).

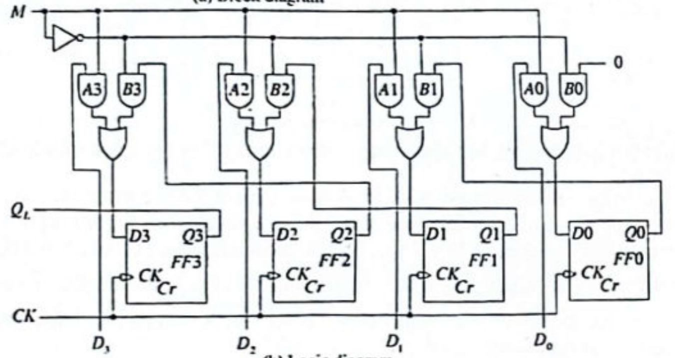
11.6.2 Parallel-In Serial-Out Left-Shift Register

The block diagram and a logic diagram for a 4-bit parallel-in serial-out left-shift resister with synchronous loading are shown in Figure 11.13. The register is constructed with D flip-flops. The circuit is similar to the circuit of Figure 11.11, except that it shifts data to the left.

Another block diagram and a logic diagram for a 4-bit parallel-in serial-out left-shift register with asynchronous loading are shown in Figure 11.14. The register is constructed with D flip-flops. The circuit is similar to the circuit of Figure 11.12, except that it shifts data to the left.

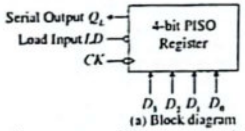

Serial Output Q_L
Mode Control M
CK
4-bit PISO Register
D_3 D_2 D_1 D_0

(a) Block diagram

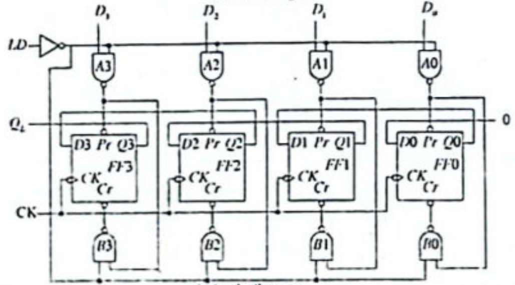

M
0
A3 B3
A2 B2
A1 B1
A0 B0
Q_L
D3 Q3 FF3 CK Cr
D2 Q2 FF2 CK Cr
D1 Q1 FF1 CK Cr
D0 Q0 FF0 CK Cr
CK
D_3
D_2
D_1
D_0

(b) Logic diagram

Figure 11.13 A 4-bit parallel-in serial-out left-shift register (synchronous loading).


Serial Output Q_L
Load Input LD
CK
4-bit PISO Register
D_3 D_2 D_1 D_0

(a) Block diagram

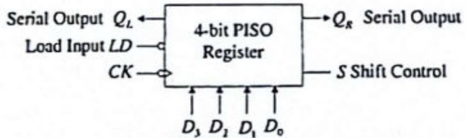

D_3
D_2
D_1
D_0
LD
A3
A2
A1
A0
Q_L
0
D3 Pr Q3 FF3 CK Cr
D2 Pr Q2 FF2 CK Cr
D1 Pr Q1 FF1 CK Cr
D0 Pr Q0 FF0 CK Cr
CK
B3
B2
B1
B0

(b) Logic diagram

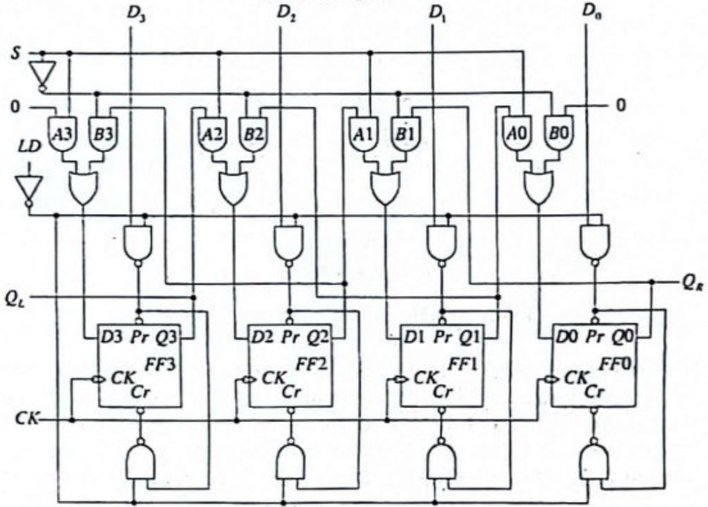
Figure 11.14 A 4-bit parallel-in serial-out left-shift register (asynchronous loading).

11.6.3 Parallel-In Serial-Out Bi-Directional Shift Register

The block diagram and a logic diagram for a 4-bit parallel-in serial-out bi-directional shift register are shown in Figure 11.15. The register is with asynchronous loading and is constructed using D flip-flops. When the load input (*LD*) is 0 the data at the parallel inputs ($D_3D_2D_1D_0$) is asynchronously loaded into the flip-flops, that is, the data is loaded when the clock is 0. During data shifting the *LD* input is set to 1. When shift control input (*S*) is 1 all *A* AND gates are enabled and the register acts as right-shift register. When *S* is 0 all *B* AND gates are enabled and the register acts as left-shift register.


Serial Output Q_L
Load Input LD
CK
4-bit PISO Register
Q_R Serial Output
S Shift Control
D_3 D_2 D_1 D_0

(a) Block diagram

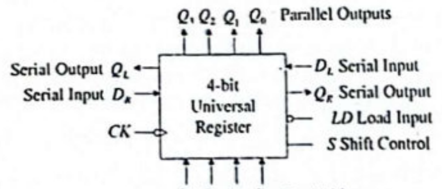

D_3
D_2
D_1
D_0
S
0
0
LD
A3 B3
A2 B2
A1 B1
A0 B0
Q_L
Q_R
D3 Pr Q3 FF3 CK Cr
D2 Pr Q2 FF2 CK Cr
D1 Pr Q1 FF1 CK Cr
D0 Pr Q0 FF0 CK Cr
CK

(b) Logic diagram

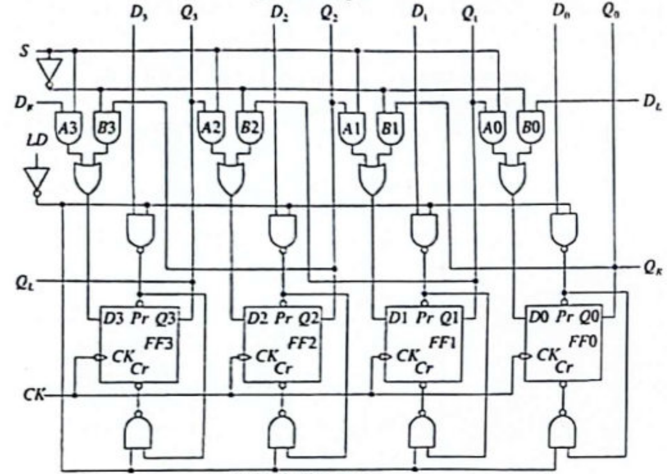
Figure 11.15 A 4-bit parallel-in serial-out bi-directional shift register.

11.7 Universal Register

The block diagram and a logic diagram for a 4-bit universal register are shown in Figure 11.16. The register is constructed using D flip-flops. The operation of the register in four modes is discussed below.


Q_3 Q_2 Q_1 Q_0 Parallel Outputs
Serial Output Q_L
Serial Input D_R
CK
4-bit Universal Register
D_L Serial Input
Q_R Serial Output
LD Load Input
S Shift Control
D_3 D_2 D_1 D_0 Parallel Inputs

(a) Block diagram


D_3
Q_3
D_2
Q_2
D_1
Q_1
D_0
Q_0
S
D_R
LD
D_L
A3
B3
A2
B2
A1
B1
A0
B0
Q_R
Q_L
D3 Pr Q3
FF3
CK Cr
D2 Pr Q2
FF2
CK Cr
D1 Pr Q1
FF1
CK Cr
D0 Pr Q0
FF0
CK Cr
CK

(b) Logic diagram

Figure 11.16 A 4-bit universal register.

Serial-in serial-out mode: When the load input (*LD*) is 1, then the register acts as a serial-in serial-out register. The direction of the shift is controlled by the shift control input (*S*). When *S* is 1 the register shifts data in the right direction, when *S* is 0 the register shifts data in the left direction.

Serial-in parallel-out mode: The operation in the serial-in parallel-out mode is the same as that in the serial-in serial-out mode, except that the outputs are taken from the parallel outputs ($Q_3Q_2Q_1Q_0$).

Parallel-in parallel-out mode: When the *LD* is 0 the data word at the parallel inputs ($D_3D_2D_1D_0$) is loaded into the register asynchronously, that is, the data is loaded when the clock is 0. After the asynchronous loading, the data is available at the parallel outputs ($Q_3Q_2Q_1Q_0$).

Parallel-in serial-out mode: The data word at the parallel inputs ($D_3D_2D_1D_0$) is asynchronously loaded into the register by setting *LD* input to 0. After asynchronous loading of data the *LD* input is set to 1 for data shifting. The direction of shift is controlled by the shift control input (*S*).

11.8 Available MSI Register ICs

Registers can be designed using discrete flip-flops as discussed in the previous sections. However, registers are available as MSI ICs. The registers available in TTL 7400 series are given in Table 11.1.

Table 11.1 Available MSI register ICs.

IC No.	Description
7491A	8-bit serial-in serial-out shift register
7494	4-bit serial/parallel-in serial-out shift register
7495	4-bit serial/parallel-in parallel-out (right-shift, left-shift) register
7496	5-bit parallel-in/parallel-out, serial-in/serial-out register
74164	8-bit serial-in parallel-out shift register
74165	8-bit serial/parallel-in serial-out shift register
74166	8-bit serial/parallel-in serial-out shift register
74194	4-bit bi-directional (universal) register
74195	4-bit serial/parallel-in serial/parallel-out register
74199	8-bit serial/parallel-in parallel-out register
74295	4-bit TRI-STATE serial/parallel-in parallel-out bi-directional register
74395	4-bit TRI-STATE cascadeble serial/parallel-in serial/parallel-out register

IC 74194 is introduced here. IC 74194 is a 4-bit bi-directional universal register. The pin assignment of IC 74194 is shown in Figure 11.17. The parallel data inputs are $D_0D_1D_2D_3$ and the parallel outputs are $Q_0Q_1Q_2Q_3$. The serial inputs are D_{SL} for left-shift and D_{SR} for right-shift. The master reset input $MR' = 0$ clears all the outputs $Q_0Q_1Q_2Q_3$. $MR' = 1$ brings the register into normal operating condition. Mode select input S_1 and S_0 defines the various operating modes of the register. $S_1 = 1$ and $S_0 = 0$ makes the register shift-left. $S_1 = 0$ and $S_0 = 1$ makes the register shift-right. $S_1 = 1$ and $S_0 = 1$ loads the parallel data inputs $D_0D_1D_2D_3$ into outputs $Q_0Q_1Q_2Q_3$. $S_1 = 0$ and $S_0 = 0$ retains the existing data in hold mode. The transition of S_1 and S_0 inputs from 1 to 0 should be made while clock input $CK = 1$. The clock (*CK*) of the register is rising-edge-triggered.

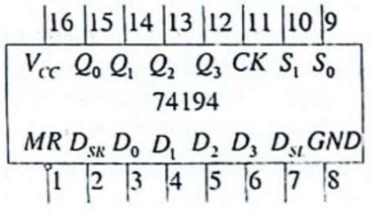

16
15
14
13
12
11
10
9
V_{CC}
Q_0
Q_1
Q_2
Q_3
CK
S_1
S_0
74194
MR
D_{SR}
D_0
D_1
D_2
D_3
D_{SL}
GND
1
2
3
4
5
6
7
8

Figure 11.17 IC 74194 4-bit bi-directional universal register.

11.9 Application of Registers

The primary uses of registers are temporary data storage and bit manipulations. Besides, registers can be used in other purposes. Some of the common applications of shift registers are discussed below.

Delay Line

A SISO register may be used to introduce time delay Δt in digital signals given by

$$\Delta t = N \times (1/f_c)$$

where N is the number of stages of the SISO register and f_c is the clock frequency. Thus, an input pulse train appears at the output delayed by Δt. The amount of delay can be controlled by the clock frequency or the number of flip-flops in the SISO register.

Serial-to-Parallel Converter

Data in the serial form can be converted into parallel form by using a SIPO register.

Parallel-to-Serial Converter

Data in the parallel form can be converted into serial form by using the PISO register.

11.10 Counters and Their Classification

A sequential circuit that goes through a prescribed sequence of states upon the application of input pulses is called a *counter*. The input pulses, called *count pulses*, may be clock pulses, or they may originate from an external source and may occur at prescribed intervals of time or at random. A counter is essentially a register that goes through a predetermined sequence of states upon the application of input pulses. The gates in a counter are connected in such a way as to produce a prescribed sequence of binary states in the register. Although counters are a special type of registers, it is common to differentiate them by giving them a special name. Counters are extensively used in the design of digital systems in general and digital computers in particular. Counters are used for counting the number of occurrences of an event and useful for generating timing variables to sequence and control the operations in a digital system.

In a counter, the sequence of states may follow a binary count sequence or any other sequence of states. Of the various sequence a counter may follow, the straight binary count sequence is the simplest and most straightforward. A counter that follows the binary count sequence is called a *binary counter*. An n-bit binary counter consists of n flip-flops and can count in binary from 0 to $2^n - 1$ and return to 0 to repeat the sequence. A counter with n flip-flops may have a binary count sequence of less than 2^n numbers. A BCD counter counts the binary count sequence from 0000 to 1001 and returns to 0000 to repeat the sequence. In general a binary counter counts in binary from 0 to $M - 1$ and returns to 0 to repeat the sequence, where $M \leq 2^n$. The M is called the *modulus* of the counter. This type of counters are called *modulo M* or $\div M$ (*divided-by-M*) counters, since these counters give a count value equal to the number of count pulses modulo M.

A $\div M$ counter may count from 0 to $M - 1$ and then to 0 to repeat the sequence, or from $M - 1$ to 0 and $M - 1$ to repeat the sequence. The counter that counts in usual count sequence of 0 to $M - 1$ is called a *binary up counter*. The counter that counts in a reverse count sequence of $M - 1$ to 0 is called a *binary down counter*.

A binary counter may be asynchronous (or ripple) counter or synchronous counter. In an *asynchronous* (or *ripple*) *counter* all the flip-flops are not clocked simultaneously, whereas in a *synchronous counter* all the flip-flops are clocked simultaneously.

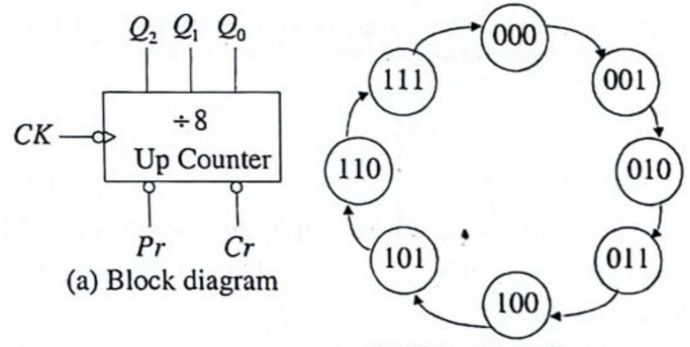
Counters may follow an arbitrary count sequence, which may not be the straight binary sequence. These counters are called *odd sequence counters*. There are some *special counters* that actually not count in a sequence but rotate a pulse in a regular fashion within the flip-flops of the counters.

11.11 Binary Asynchronous Counters

In asynchronous or ripple counter, the flip-flop output transition serves as a source for triggering other flip-flops. In other words, the CK inputs of all flip-flops (except the first) are triggered not by the incoming pulses but rather by the transition that occurs in other flip-flops.

11.11.1 Binary Asynchronous Up Counter

Block diagram and transition diagram of a ÷8 up counter are shown in Figure 11.18. As seen from the binary states indicated inside the circles, the flip-flop outputs repeat the binary count sequence with a return to 000 after 111. The directed lines between circles are not marked with input-output values as in other transition diagrams, since there is no external input and the outputs of the flip-flops are taken as the output of the counter. The count sequence of a ÷8 up counter is shown in Figure 11.19.



(a) Block diagram

(b) Transition diagram

Figure 11.18 Block diagram and transition diagram of a ÷8 up counter.

Count Sequence

Q_2	Q_1	Q_0
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1
0	0	0

Figure 11.19 Count sequence of a ÷8 up counter.

For designing a ÷8 asynchronous up counter we observe the transition of each flip-flop output in Figure 11.19. The output $Q0$ of the least-significant flip-flop changes for every clock pulse. This can be achieved by using a falling edge triggered T flip-flop *FF*0 with $T0 = 1$. The output $Q1$ makes a transition (from 0 to 1 or 1 to 0) whenever $Q0$ changes from 1 to 0. Therefore, if $Q0$ is connected to the clock input of next falling edge triggered flip-flop *FF*1 with $T1 = 1$, $Q1$ will change whenever $Q0$ goes from 1 to 0 (falling edge of clock pulse). Similarly, $Q2$ makes a transition whenever $Q1$ goes from 1 to 0 and this can be achieved by connecting $Q1$ to the clock input of the most-significant falling edge triggered flip-flop *FF*2 with $T2 = 1$. The resulting circuit is shown in Figure 11.20 and the waveforms of the outputs of the flip-flops are shown in Figure 11.21.

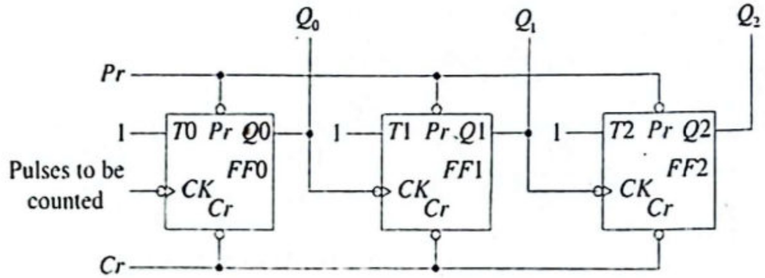


Figure 11.20 Logic diagram of a ÷8 asynchronous up counter.

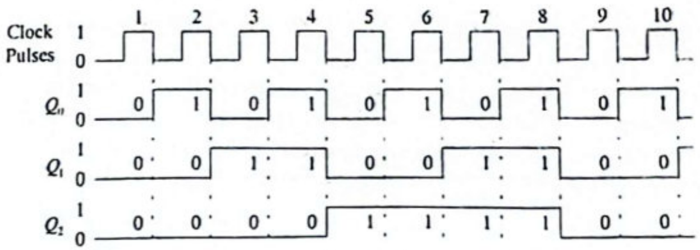
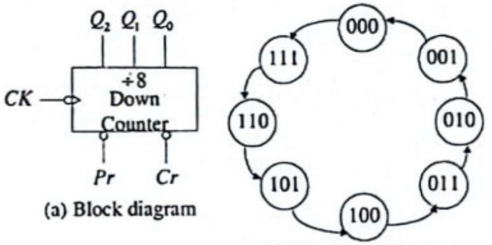


Figure 11.21 Output waveforms of counter of Figure 11.19.

11.11.2 Binary Asynchronous Down Counter

Block diagram and transition diagram of a ÷8 down counter are shown in Figure 11.22. The count sequence is shown in Figure 11.23.



(a) Block diagram

(b) Transition diagram

Figure 11.22 Block diagram and transition diagram of a ÷8 down counter.

Count Sequence

Q_2	Q_1	Q_0
1	1	1
1	1	0
1	0	1
1	0	0
0	1	1
0	1	0
0	0	1
0	0	0
1	1	1

Figure 11.23 Count sequence of a ÷8 down counter.

From Figure 11.23 we observe that the output $Q0$ of the least-significant flip-flop changes for every clock pulse. This can be achieved by using a falling edge triggered T flip-flop *FF*0 with $T0 = 1$. The output $Q1$ makes a transition (from 0 to 1 or 1 to 0) whenever $Q0$ changes from 0 to 1, that is, whenever $Q0'$ changes from 1 to 0. Therefore, if $Q0'$ is connected to the clock input of the next falling edge triggered T flip-flop *FF*1 with $T1 = 1$, $Q1$ will change whenever $Q0'$ goes from 1 to 0 (falling edge of clock pulse). Similarly, $Q2$ makes a transition whenever $Q1$ goes from 0 to 1, that is, whenever $Q1'$ goes from 1 to 0 and this can be achieved by connecting $Q1'$ to the clock input of the most-significant falling edge triggered T flip-flop *FF*2 with $T2 = 1$. The resulting circuit is shown in Figure 11.24 and the waveforms of the outputs of the flip-flops are shown in Figure 11.25.

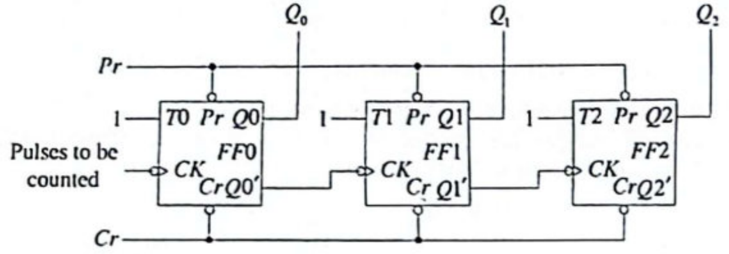


Figure 11.24 Logic diagram of a ÷8 asynchronous down counter.

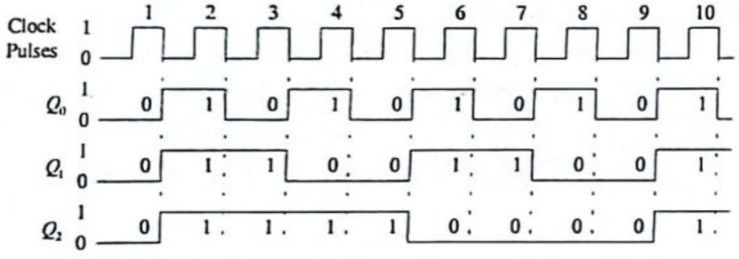
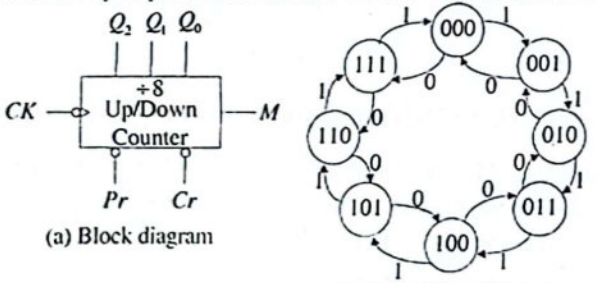


Figure 11.25 Output waveforms of counter of Figure 11.24.

11.11.3 Binary Asynchronous Up/Down Counter

Block diagram and transition diagram of a ÷8 up/down counter are shown in Figure 11.26. As seen from the block diagram and the transition diagram, a ÷8 up/down counter has an external input *M* for controlling the count sequence. When *M* is 1 the counter counts in the up sequence and when *M* is 0 the counter counts in the down sequence.



(a) Block diagram

(b) Transition diagram

Figure 11.26 Block diagram and transition diagram of a ÷8 up/down counter.

The logic diagram for a ÷8 asynchronous up/down counter is shown in Figure 11.27. In the logic diagram of Figure 11.27, when *M* is 1 all *A* AND gates are enabled and all *B* AND gates are disabled. Therefore, $Q0$ appears at the clock input of *FF*1 and $Q1$ appears at the clock input of *FF*2 and the circuit counts in the up sequence. When *M* is 0 all *B* AND gates are enabled and all *A* AND gates are disabled. Therefore, $Q0'$ appears at the clock input of *FF*1 and $Q1'$ appears at the clock input of *FF*2 and the circuit counts in the down sequence.

In a similar manner a $\div 2^n$ asynchronous up, down or up/down counter can be designed for any value of *n*.

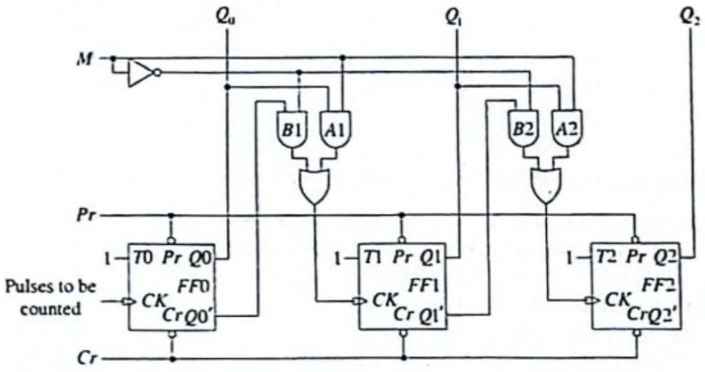


Figure 11.27 Logic diagram of a ÷8 asynchronous up/down counter.

11.11.4 Binary Asynchronous Presettable Counter

A counter that can be preset to any desired starting count is referred to as ***presettable*** counter. An asynchronous counter can be designed to be presettable. This presetting operation is referred to as *loading* the counter. The asynchronous loading technique of registers discussed in Section 11.5 can be used for asynchronous loading of an asynchronous counter. The circuit diagram of a ÷8 up asynchronous presettable counter is shown in Figure 11.28. In Figure 11.28 D_0, D_1, and D_2 are the parallel data inputs and *Ld* is the asynchronous load control input. When the *Ld* input is 1 the counter counts in its normal count sequence. When the *Ld* input is 0 and the clock *CK* is also 0, then the data present at the parallel inputs are loaded asynchronously to all flip-flops of the counter.

11.11.5 Binary Asynchronous Counter with Modulus Less Than 2^n

The counters discussed above are modulo 2^n counter. If it is desired to have a modulo *M* (where $M < 2^n$) counter, the modulo 2^n counter can easily be converted into a modulo *M* counter. In case of constructing a ÷*M* up counter, the $\div 2^n$ up counter is required to be reset (that is, the normal counting is to be terminated) at the end of the *M*th clock pulse. This can be achieved by generating a 0 signal immediately after the *M*th pulse and applying it to the clear input of all the flip-flops.

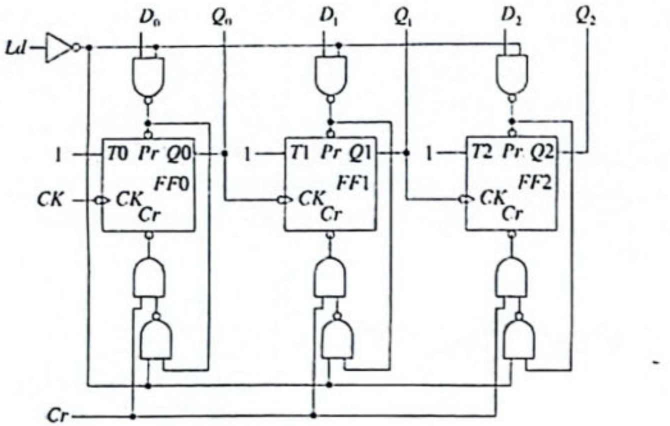
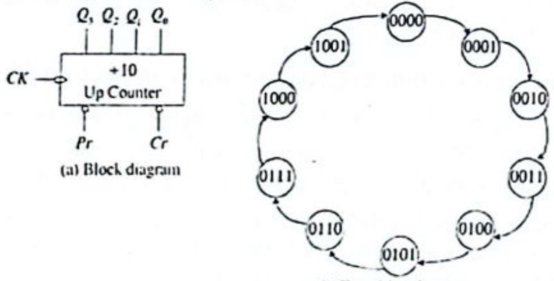


Figure 11.28 Logic diagram of a $\div 8$ up asynchronous presettable counter.

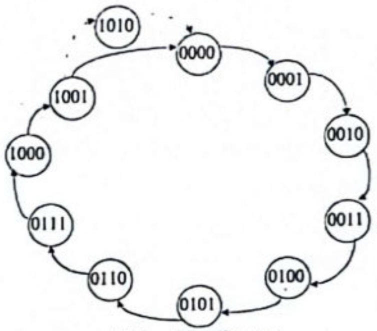
Suppose that we require a $\div 10$ asynchronous up counter (also referred to as BCD asynchronous counter). Block diagram and transition diagram of a $\div 10$ up counter are shown in Figure 11.29. From the transition diagram of Figure 11.29 we observe that after counting 1001 the next count will be 0000. But in a $\div 16$ up counter the next count will be 1010. For converting a $\div 16$ up counter into a $\div 10$ up counter, the count 1010 is to be decoded to 0 and to be applied to *Cr* input of the counter to reset the count to 0000. The required conversion is shown in Figure 11.30(a) and the converted circuit is shown in Figure 11.30(b). When the count of a $\div 16$ asynchronous up counter is between 0000 to 1001 inclusive the output of the NAND gate in Figure 11.30(b) will always be 1. When the count is 1010 the output of the NAND gate will be 0 and the output of all the flip-flops of $\div 16$ asynchronous up counter will be asynchronously reset to 0 resulting in a count of 0000. Although the counter does go to the 1010 state, it remains there for only few nanoseconds before it recycles to 0000. Thus, we can essentially say that the converted counter in Figure 11.30(b) counts from 0000 to 1001 and then recycles to 0000. Following the same procedure any $\div 2^n$ asynchronous up counter can be converted into a $\div M$ (where $M < 2^n$) asynchronous up counter.



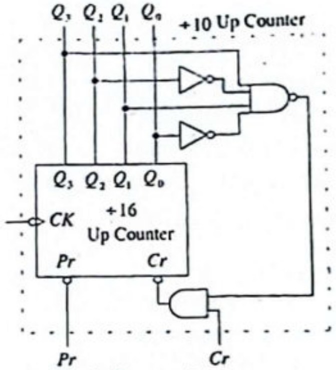
(a) Block diagram

(b) Transition diagram

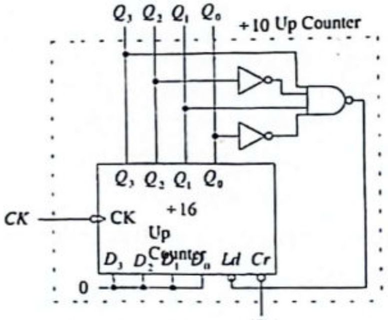
Figure 11.29 Block diagram and transition diagram of a $\div 10$ asynchronous up counter.



(a) Transition diagram



(b) Circuit diagram using up counter



(c) Circuit diagram using up presettable counter

Figure 11.30 Conversion of a $\div 16$ asynchronous up counter into a $\div 10$ asynchronous up counter.

A $\div 2^n$ up asynchronous presettable counter can also be converted into a $\div M$ up asynchronous counter. In this case the parallel inputs are set to 0. The *M*th count is decoded to 0 and applied to the asynchronous load input *Ld*. When the counter counts *M* it is decoded to 0 and the *Ld* input becomes 0. Immediately the parallel inputs are loaded into the counter resulting in a count of 0. The conversion of a $\div 16$ up asynchronous presettable counter into a $\div 10$ up asynchronous counter is shown in Figure 11.30(c).

A $\div 2^n$ down asynchronous presettable counter can be converted into a $\div M$ (where $M < 2^n$) down asynchronous counter. Block diagram and transition diagram of a $\div 10$ down counter is shown in Figure 11.31. From the transition diagram of Figure 11.31 we observe that after counting 0000 the next count will be 1001. But in a $\div 16$ down counter the next count will be 1111. For converting a $\div 16$ down asynchronous presettable counter into a $\div 10$ down asynchronous counter the count 1111 is to be decoded to 0 and to be applied to the asynchronous load input *Ld*. The parallel inputs are to be set to 1001. The required conversion is shown in Figure 11.32. When the count sequence of the $\div 16$ down asynchronous counter is

1001 to 0000 inclusive the output of the NAND gate will always be 1. When the count is 1111 the output of the NAND gate will be 0 and the parallel input 1001 will be asynchronously loaded into the counter resulting a count of 1001. Thus the counter counts from 1001 to 0000 and then recycles to 1001. Following the same procedure any $\div 2^n$ down asynchronous presettable counter can be converted into a $\div M$ (where $M < 2^n$) down asynchronous counter.

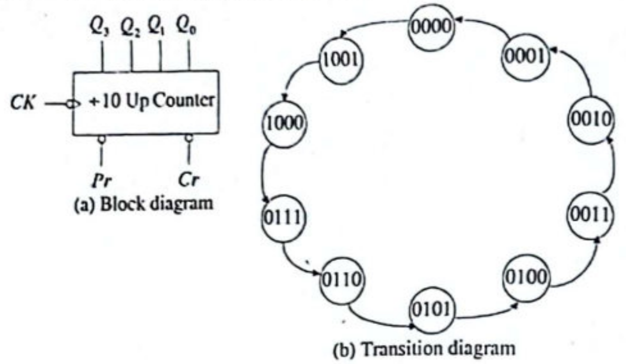

Q_3 Q_2 Q_1 Q_0
CK
÷10 Up Counter
Pr
Cr
(a) Block diagram
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
(b) Transition diagram

Figure 11.31 Block diagram and transition diagram of a ÷10 down counter.

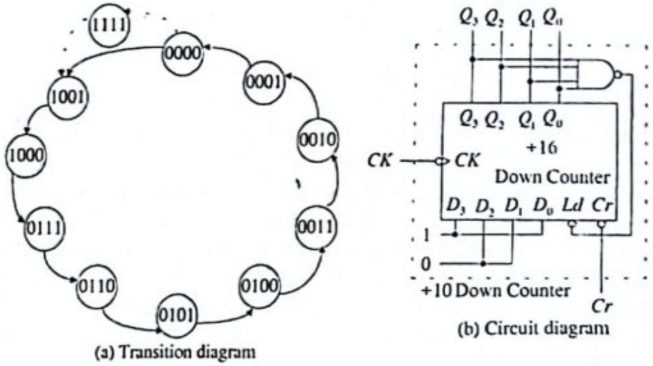

1111
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
(a) Transition diagram
Q_3 Q_2 Q_1 Q_0
÷16
CK
Down Counter
D_3 D_2 D_1 D_0 Ld Cr
1
0
÷10 Down Counter
Cr
(b) Circuit diagram

Figure 11.32 Conversion of a ÷16 down asynchronous presettable counter into a ÷10 down asynchronous counter.

A $\div 2^n$ up asynchronous presettable counter can be made to count from M to N (where $0 \le M < N < 2^n$). For example, a ÷8 up asynchronous presettable counter can be made to count from 3 to 6 and then to 3 to continue the count sequence. The required conversion is shown in Figure 11.33. The count 111 is decoded to 0 and applied to the *Ld* input. Suppose that a 0 is applied to the *Ld* input during power on, then the counter will be asynchronously loaded to an initial value of 011. After initial asynchronous loading the *Ld* input is set to 1 for normal

---

operation. When the count sequence of the ÷8 up asynchronous counter is 011 to 110 inclusive the output of the NAND gate will always be 1. When the count is 111 the output of the NAND gate will be 0 and the parallel input 011 will be asynchronously loaded into the counter resulting in a count of 011. Thus the counter counts from 011 to 110 and then recycles to 011.

A $\div 2^n$ down asynchronous presettable counter can also be made to count from N to M (where $0 \le M < N < 2^n$). For example, a ÷8 down asynchronous presettable counter can be made to count from 6 to 3 and then to 6 to continue the count sequence. The required conversion is shown in Figure 11.34. The count 010 is decoded to 0 and applied to the *Ld* input. Suppose that the counter is asynchronously loaded to initial value 110 during power on. When the count sequence of the ÷8 down asynchronous counter is 110 to 011 inclusive the output of the NAND gate will always be 1. When the count is 010 the output of the NAND gate will be 0 and the parallel input 110 will be asynchronously loaded into the counter resulting in a count of 110. Thus the counter counts 110 to 011 and the recycles to 110.

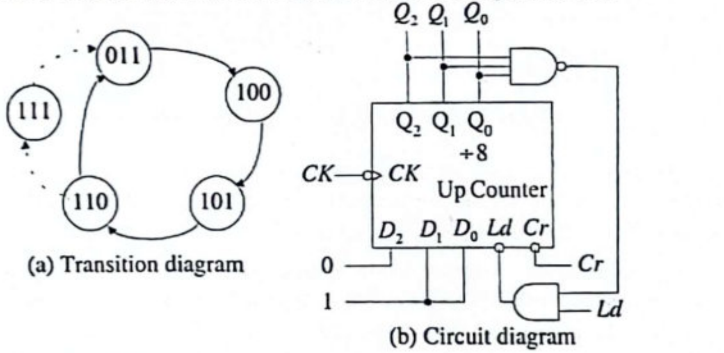

011
100
101
110
111
(a) Transition diagram
Q_2 Q_1 Q_0
Q_2 Q_1 Q_0
÷8
CK
CK
Up Counter
D_2 D_1 D_0 Ld Cr
0
1
Cr
Ld
(b) Circuit diagram

Figure 11.33 Conversion of a ÷8 up asynchronous presettable counter to count from 3 to 6.

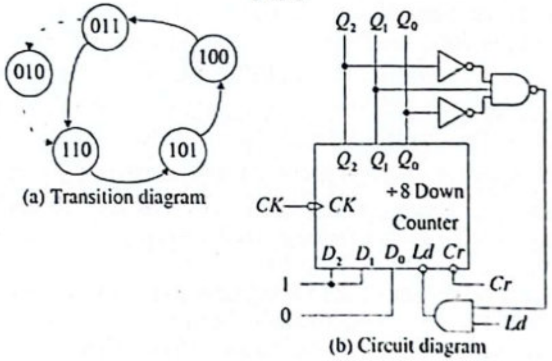

011
100
101
110
010
(a) Transition diagram
Q_2 Q_1 Q_0
Q_2 Q_1 Q_0
÷8 Down
CK
CK
Counter
D_2 D_1 D_0 Ld Cr
1
0
Cr
Ld
(b) Circuit diagram

Figure 11.34 Conversion of a ÷8 down asynchronous presettable counter to count from 6 to 3.

11.11.6 Cascading Of Binary Asynchronous Counters

A $\div 2^{m+n}$ asynchronous counter can be made by cascading one $\div 2^m$ and one $\div 2^n$ asynchronous counter. In case of cascading asynchronous up counters, the most-significant output of the preceding counter is connected to the clock input of the succeeding counter. The *Pr* and *Cr* input of all counters are connected together. In case of cascading asynchronous down counter, the most-significant output of the preceding counter is inverted and then connected to the clock input of the succeeding counter. The *Pr* and *Cr* inputs of all counters are also connected together. This procedure is illustrated in Figure 11.35.

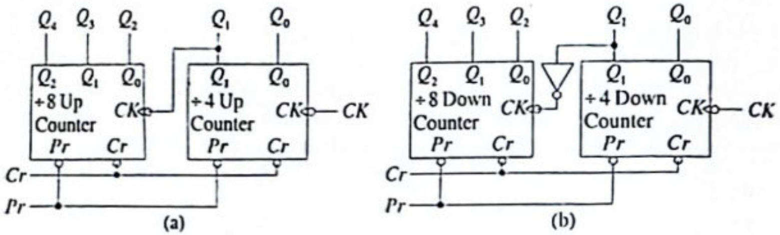


Figure 11.35 Cascading of asynchronous counters.

In Figure 11.35(a) a ÷8 and a ÷4 up asynchronous counter are cascaded to construct a ÷32 up asynchronous counter. Suppose that the initial count of the combined counter is $Q_4Q_3Q_2Q_1Q_0 = 00000$. For the next three clock pulses the ÷4 asynchronous up counter counts the sequence 01, 10, 11 and the combined count sequence becomes 00001, 00010, 00011 respectively. At the next clock the ÷4 asynchronous up counter counts 00 and the output Q_1 goes from 1 to 0, the ÷8 asynchronous up counter then counts 001 and the combined count becomes 00100. In this way the combined counter acts as a ÷32 asynchronous up counter.

In Figure 11.35(b) a ÷8 and a ÷4 asynchronous down counter are cascaded to construct a ÷32 asynchronous down counter. Suppose that the initial count of the combined counter is $Q_4Q_3Q_2Q_1Q_0 = 11111$. For the next three clock pulses the ÷4 asynchronous down counter counts the sequence 10, 01, 00 and the combined count sequence becomes 11110, 11101, 11100 respectively. At the next clock pulse the ÷4 asynchronous down counter counts 11 and the output Q_1 goes from 0 to 1, the ÷8 asynchronous down counter then counts 110 and the combined count becomes 11011. In this way the combined counter acts as a ÷32 asynchronous down counter.

In a similar fashion an *n*-digit BCD asynchronous counter can be constructed by cascading *n* BCD asynchronous counters. This procedure is illustrated in Figure 11.36 for cascading two BCD asynchronous counters. In Figure 11.36 two BCD asynchronous up counters are cascaded to construct a 2-digit BCD asynchronous up counter. Suppose that the initial count of the combined counter is

---

$Q_7Q_6Q_5Q_4Q_3Q_2Q_1Q_0 = 00000000$ (decimal 00). For the next nine clock pulses the BCD asynchronous up counter *C*0 counts the sequence 0001, 0010, 0011, 0100, 0101, 01100, 0111, 1000, 1001 (decimal 1, 2, 3, 4, 5, 6, 7, 8, 9) and the combined counts become 01, 02, 03, 04, 05, 06, 07, 08, 09 in decimal, respectively. At the next clock the BCD asynchronous up counter *C*0 counts 0000 (decimal 0) and the output Q_3 goes from 1 to 0, the BCD asynchronous up counter *C*1 then counts 0001 (decimal 1) and the combined count becomes 10 in decimal. In this way the combined counter acts as a 2-digit BCD asynchronous up counter.

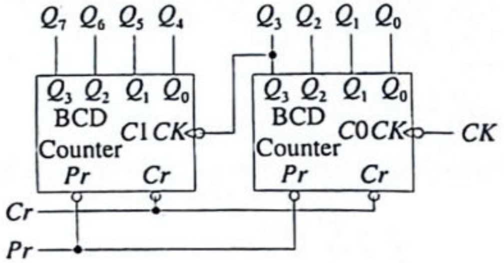


Figure 11.36 Cascading of BCD counters.

11.12 Binary Synchronous Counters

Synchronous counters are distinguished from asynchronous (or ripple) counters in that clock pulses are applied to the *CK* inputs of all flip-flops. The common pulse triggers all the flip-flops simultaneously, rather than one at a time in succession as in an asynchronous counter. The decision whether a flip-flop is to be complemented or not is determined from the values of the flip-flop inputs at the time of the pulse.

11.12.1 Binary Synchronous Up Counter

Block diagram and transition diagram of a ÷8 up counter are shown in Figure 11.18. A ÷8 up synchronous counter can be designed using the design method discussed in Section 10.5. Binary counters are most efficiently constructed using T flip-flops (though any type of flip-flops can be used). The excitation table for a ÷8 up synchronous counter using T flip-flops is shown in Table 11.2.

Table 11.2 Excitation table of a ÷8 up synchronous counter.

Present state $Q2_nQ1_nQ0_n$	Next state $Q2_{n+1}Q1_{n+1}Q0_{n+1}$	Flip-flop inputs $T2T1T0$
000	001	001
001	010	011
010	011	001
011	100	111
100	101	001
101	110	011
110	111	001
111	000	111

The simplified Boolean functions for the flip-flop inputs are derived using the K-maps of Figure 11.37, which gives

$$T2 = Q1_n Q0_n$$
$$T1 = Q0_n$$
$$T0 = 1$$

The logic diagram of a ÷8 up synchronous counter is drawn in Figure 11.38.

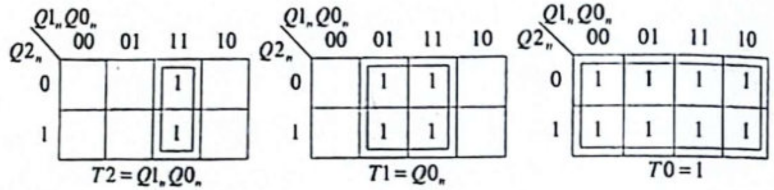

Q1nQ0n
Q2n
00 01 11 10
0
1
T2 = Q1nQ0n
T1 = Q0n
T0 = 1

Figure 11.37 K-maps for excitation equations for a ÷8 up synchronous counter using T flip-flops.

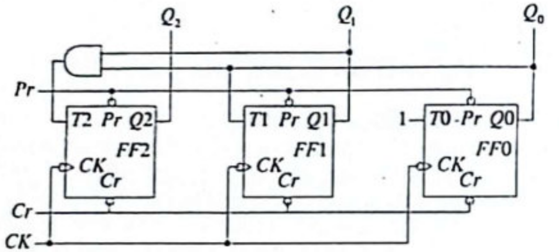

Q2
Q1
Q0
Pr
T2 Pr Q2
FF2
CK
Cr
T1 Pr Q1
FF1
CK
Cr
1
T0 Pr Q0
FF0
CK
Cr
Cr
CK

Figure 11.38 Logic diagram of a ÷8 up synchronous counter.

11.12.2 Binary Synchronous Down Counter

Block diagram and transition diagram of a ÷8 down counter are shown in Figure 11.22. The excitation table for a ÷8 down synchronous counter using T flip-flops is shown in Table 11.3.

Table 11.3 Excitation table of a ÷8 down synchronous counter using T flip-flops.

Present state $Q2_n Q1_n Q0_n$	Next state $Q2_{n+1} Q1_{n+1} Q0_{n+1}$	Flip-flop inputs $T2T1T0$
000	111	111
001	000	001
010	001	011
011	010	001
100	011	111
101	100	001
110	101	011
111	110	001

The simplified Boolean functions for the flip-flop inputs are derived using the K-maps of Figure 11.39, which give

$$T2 = Q1'_n Q0'_n$$
$$T1 = Q0'_n$$
$$T0 = 1$$

The logic diagram of a ÷8 down synchronous counter is drawn in Figure 11.40.

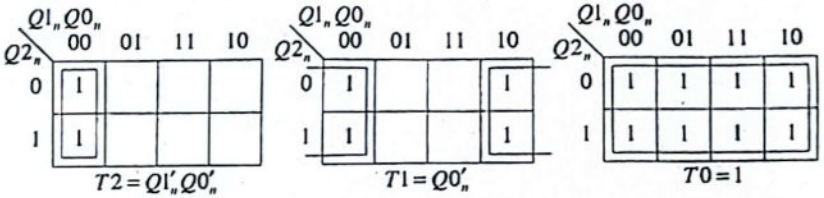

Q1nQ0n
Q2n
00 01 11 10
0
1
T2 = Q1'nQ0'n
T1 = Q0'n
T0 = 1

Figure 11.39 K-maps for excitation equations of a ÷8 down synchronous counter using T flip-flops.

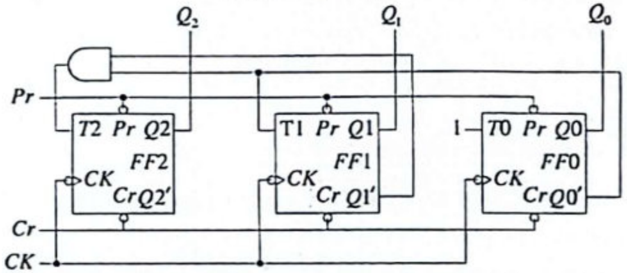

Q2
Q1
Q0
Pr
T2 Pr Q2
FF2
CK
Cr Q2'
T1 Pr Q1
FF1
CK
Cr Q1'
1
T0 Pr Q0
FF0
CK
Cr Q0'
Cr
CK

Figure 11.40 Logic diagram of a ÷8 down synchronous counter.

11.12.3 Binary Synchronous Up/Down Counter

Block diagram and transition diagram of a ÷8 up/down counter are shown in Figure 11.26. The excitation table for a ÷8 up/down synchronous counter using T flip-flops is shown in Table 11.4.

Table 11.4 Excitation table of a ÷8 up/down synchronous counter using T flip-flops.

Present state $Q2_n Q1_n Q0_n$	Next state $Q2_{n+1} Q1_{n+1} Q0_{n+1}$		Flip-flop inputs $T2T1T0$	
	$M = 0$	$M = 1$	$M = 0$	$M = 1$
000	111	001	111	001
001	000	010	001	011
010	001	011	011	001
011	010	100	001	111
100	011	101	111	001
101	100	110	001	011
110	101	111	011	001
111	110	000	001	111

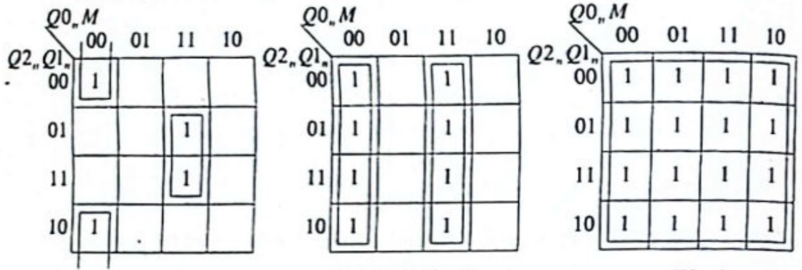
The simplified Boolean functions for the flip-flop inputs are derived using the K-maps of Figure 11.41, which give

$$T2 = Q1'_n Q0'_n M' + Q1_n Q0_n M$$

$$T2 = Q0'_n M' + Q0_n M$$

$$T0 = 1$$

The logic diagram of a ÷8 up/down synchronous counter is drawn in Figure 11.42.



$T2 = Q1'_n Q0'_n M' + Q1_n Q0_n M$ $T2 = Q0'_n M' + Q0_n M$ $T0 = 1$

Figure 11.41 K-maps for excitation equations of a ÷8 up/down synchronous counter using T flip-flops.

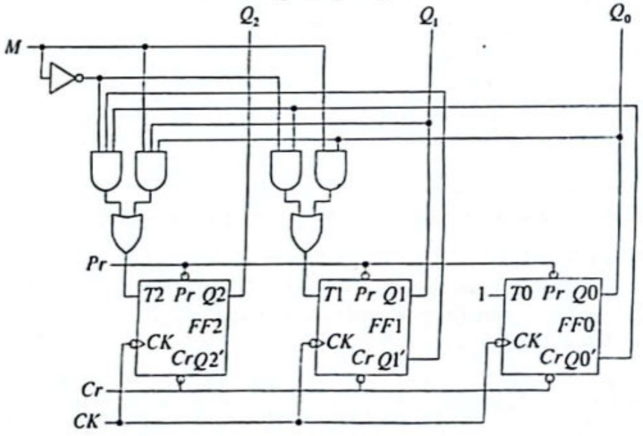


Figure 11.42 Logic diagram of a ÷8 up/down synchronous counter.

In a similar manner a $\div 2^n$ synchronous up, down or up/down counter can be designed for any value of *n*.

11.12.4 Binary Synchronous Presettable Counters

Any synchronous counter can be designed to be presettable using the asynchronous loading technique discussed in Section 11.2. The logic diagram of ÷8 up synchronous counter shown in Figure 11.38 is converted for making presettable counter and is shown in Figure 11.43. In Figure 11.43, when the *Ld* input is 1 the counter counts in its normal count sequence. When the *Ld* input is 0 and the *CK* is also 0, then the data present in the parallel inputs $D_2 D_1 D_0$ are loaded asynchronously to all flip-flops of the counter.

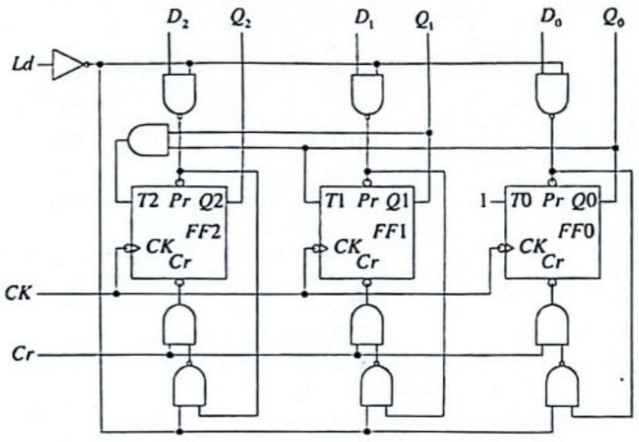


Figure 11.43 Logic diagram of a ÷8 up synchronous presettable counter.

11.12.5 Binary Synchronous Counter With Modulus Less Than 2^n

Any $\div 2^n$ synchronous counter with *Cr* input or/and presettable load inputs can be converted into a ÷*M* (where $M < 2^n$) synchronous counter using the method discussed in Section 11.11.5. For example, a ÷16 up synchronous counter can be converted into a ÷10 up synchronous counter as in Figure 11.30. However, a ÷*M* synchronous counter can be designed using the standard design procedure of synchronous sequential circuits discussed in Section 10.5.

Block diagram and transition diagram of a ÷10 up counter are shown in Figure 11.29. The excitation table for a ÷10 up counter using T flip-flops is shown in Table 11.5.

The simplified Boolean functions for the flip-flop inputs are derived using the K-maps of Figure 11.44. In the K-maps the unused states 1010 to1111 are taken as don't-care conditions. The simplified excitation equations are

$$T3 = Q3_n Q0_n + Q2_n Q1_n Q0_n$$

$$T2 = Q1_n Q0_n$$

$$T1 = Q3'_n Q0_n$$

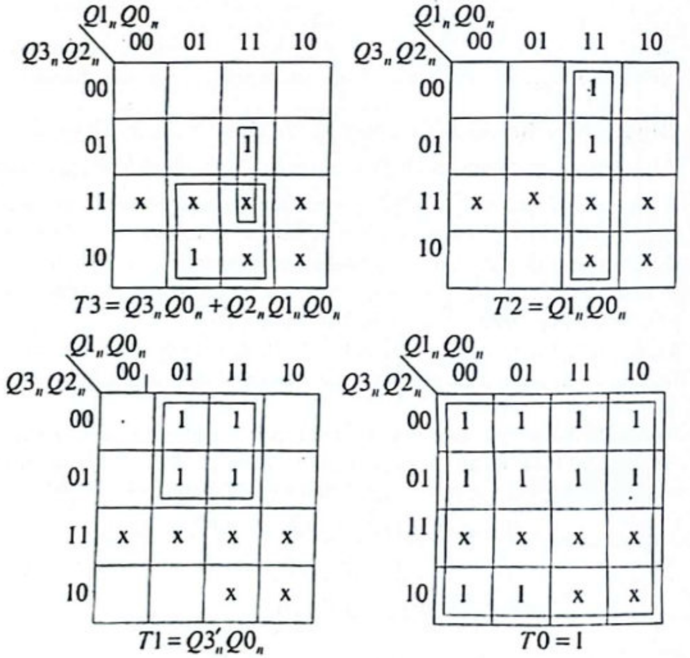
$$T0 = 1$$

The logic diagram of a ÷10 up synchronous counter is drawn in Figure 11.45.

In a similar fashion any ÷M (where $M < 2^n$) synchronous up, down or up/down counter can be designed.

Table 11.5 Excitation table of a ÷10 up synchronous counter using T flip-flops.

Present state $Q3_n Q2_n Q1_n Q0_n$	Next state $Q3_{n+1} Q2_{n+1} Q1_{n+1} Q0_{n+1}$	Flip-flop inputs $T3T2T1T0$
0000	0001	0001
0001	0010	0011
0010	0011	0001
0011	0100	0111
0100	0101	0001
0101	0110	0011
0110	0111	0001
0111	1000	1111
1000	1001	0001
1001	0000	1001



$T3 = Q3_nQ0_n + Q2_nQ1_nQ0_n$

$T2 = Q1_nQ0_n$

$T1 = Q3'_nQ0_n$

$T0 = 1$

Figure 11.44 K-maps for excitation equations of a ÷10 up synchronous counter using T flip-flops.

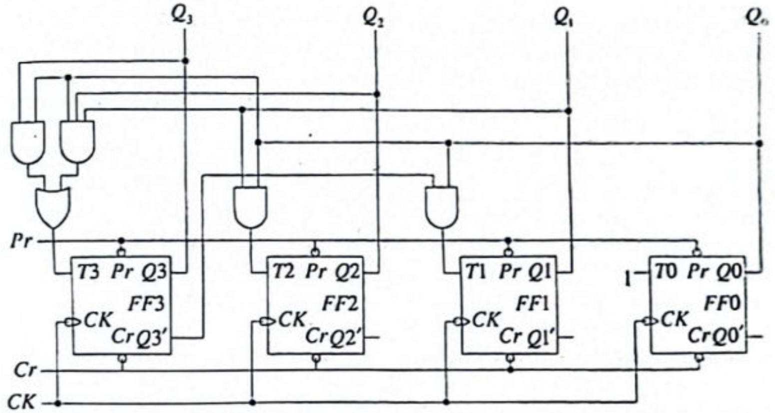


Figure 11.45 Logic diagram of a ÷10 up synchronous counter.

11.13 Odd Sequence Counter

Odd sequence counters follow arbitrary count sequences, which may not be the straight binary sequence. Odd sequence counters are designed using the design method of sequential circuits discussed in Section 10.5. For example, a counter may count the sequence 0-1-3-7-6-4 and again 0 to repeat the sequence. The transition diagram of the counter is shown in Figure 11.46. The excitation table for the counter using J-K flip-flop is shown in Table 11.6.

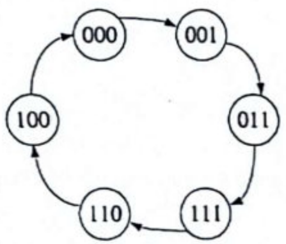


Figure 11.46 Transition diagram of an odd sequence counter.

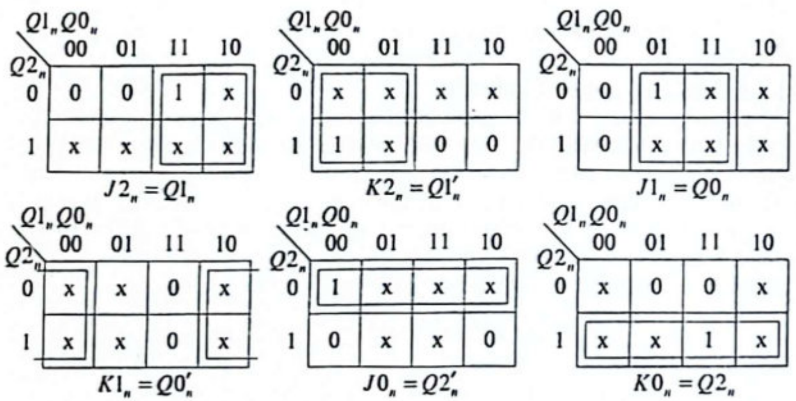
Table 11.6 Excitation table of the odd sequence counter of Figure 11.46 using J-K flip-flops.

Present state $Q2_n Q1_n Q0_n$	Next state $Q2_{n+1} Q1_{n+1} Q0_{n+1}$	Flip-flop inputs $J2K2$ $J1K1$ $J0K0$
000	001	0x 0x 1x
001	011	0x 1x x0
011	111	1x x0 x0
111	110	x0 x0 x1
110	100	x0 x1 0x
100	000	x1 0x 0x

The simplified Boolean functions for the flip-flop inputs are derived using the K-maps of Figure 11.47. In the K-maps of Figure 11.47 the unused states 010 and 101 are considered as don't care. The simplified flip-flop inputs are as below

$$J2_n = Q1_n$$
$$K2_n = Q1'_n$$
$$J1_n = Q0_n$$
$$K1_n = Q0'_n$$
$$J0_n = Q2'_n$$
$$K0_n = Q2_n$$

The logic diagram of the odd sequence counter is shown in Figure 11.48.



$Q2_n$ \ $Q1_nQ0_n$	00	01	11	10
0	0	0	1	x
1	x	x	x	x

$J2_n = Q1_n$

$Q2_n$ \ $Q1_nQ0_n$	00	01	11	10
0	x	x	x	x
1	1	x	0	0

$K2_n = Q1'_n$

$Q2_n$ \ $Q1_nQ0_n$	00	01	11	10
0	0	1	x	x
1	0	x	x	x

$J1_n = Q0_n$

$Q2_n$ \ $Q1_nQ0_n$	00	01	11	10
0	x	x	0	x
1	x	x	0	x

$K1_n = Q0'_n$

$Q2_n$ \ $Q1_nQ0_n$	00	01	11	10
0	1	x	x	x
1	0	x	x	0

$J0_n = Q2'_n$

$Q2_n$ \ $Q1_nQ0_n$	00	01	11	10
0	x	0	0	x
1	x	x	1	x

$K0_n = Q2_n$

Figure 11.47 K-maps for excitation equations of the odd sequence counter of Figure 11.46.

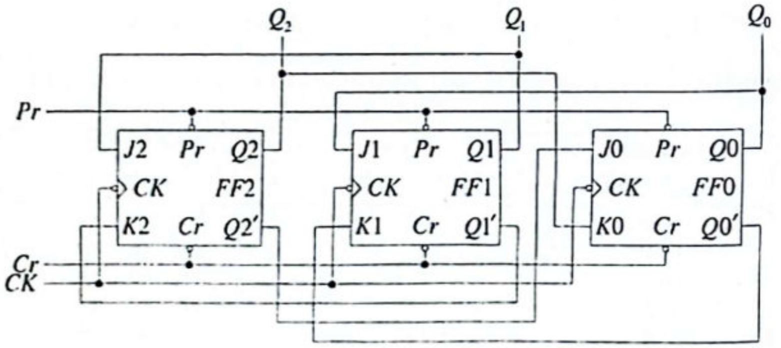


Figure 11.48 Logic diagram of the odd sequence counter of Figure 11.46.

11.14 Special Counters

There are some special counters that actually not count in a sequence but rotate a pulse in a regular fashion within the flip-flops of the counters. In this section such two counters are introduced.

11.14.1 Ring Counter

The logic diagram of a *ring counter* is shown in Figure 11.49. The flip-flop output $Q0$ is connected back to the flip-flop input $D3$. If a pulse is injected by asynchronous loading of 1000, this 1 circulates around the circuit when the clock pulses are applied. The waveforms at the outputs $Q3Q2Q1Q0$ are shown in Figure 11.50. Each flip-flop is in the 1 state once every four clock pulses. Each output becomes a 1 after the negative-edge transition of a clock pulse and remains 1 during the next clock pulse. The outputs are sequential non-overlapping pulses, which are useful for control-state counters, for stepper motor, which require sequential pulses to rotate it from one position to the next, etc. This counter can also be used for counting the number of pulses. The number of pulses counted is read by noting which flip-flop is in 1 state.

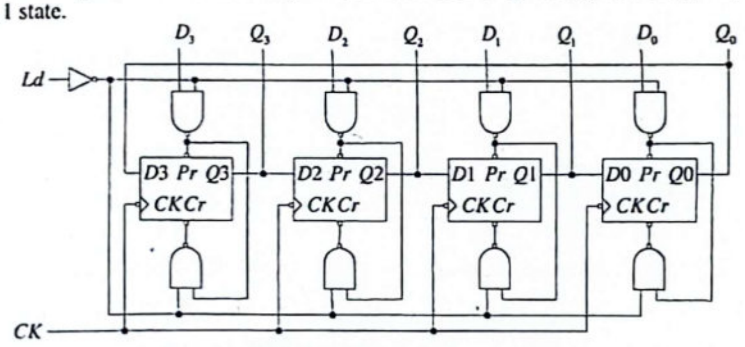


Figure 11.49 Logic diagram of ring counter.

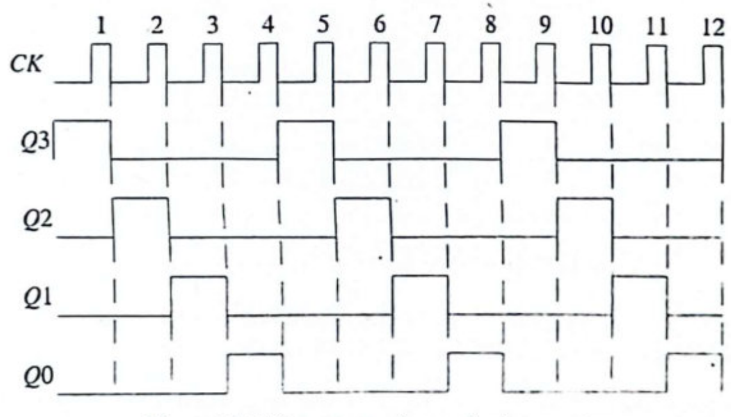


Figure 11.50 Output waveforms of a ring counter.

11.14.2 Twisted-Ring Counter

A k-bit ring counter circulates a single bit among the flip-flops to provide k distinguishable states. The number of states can be doubled if the flip-flop output $Q0'$ is connected back to flip-flop input $D3$. Such a counter is known as *twisted-ring counter* (or *Johnson* or *Moebius counter*). The logic diagram of twisted-ring counter is shown in Figure 11.51. If the clock pulses are applied after asynchronous loading of 0000, square wave is obtained at $Q3Q2Q1Q0$ outputs as shown in Figure 11.52. Each flip-flop shifts its contents once to the right with every clock pulse, and at the same time, the complement value of the $Q0$ flip-flop is transferred into the $Q3$ flip-flop. Starting from a 0000 state, each shift operation inserts 1s from the left until the flip-flops are filled with all 1s. In the following sequence 0s are inserted from the left until the flip-flops are again filled with all 0s. In this manner the twisted-ring counter goes through a sequence of eight states. In general a k-bit twisted-ring counter will go through a sequence of $2k$ states.

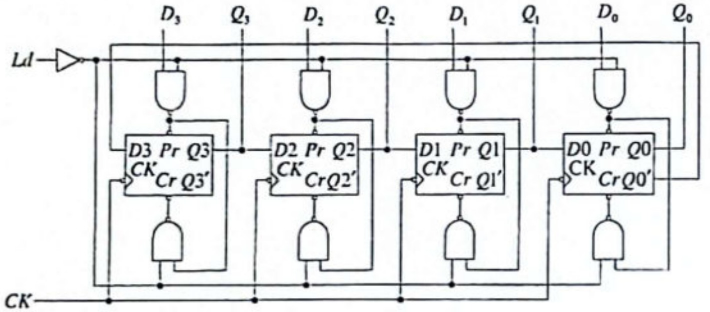


Figure 11.51 Logic diagram of a twisted-ring counter.

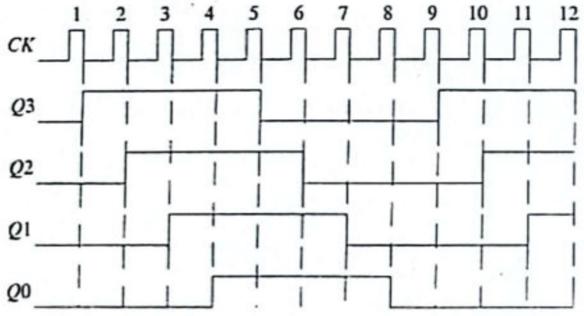


Figure 11.52 Output waveforms of a twisted-ring counter.

Similar to ring counter sequence, the twisted-ring counter sequence is also useful for control-state counter. It is also useful for the generation of multiphase clock.

The $2k$ states of a k-bit twisted-ring counter are decoded using $2k$ two-input AND gates as shown in Table 11.7. The decoding is done as follows:

- The all 0s state is decoded by taking the complement of the two extreme flip-flop outputs.
- The all 1s state is decoded by taking the normal outputs of the two extreme flip-flops.
- All other states are decoded from an adjacent 10 or 01 pattern in the sequence.

Table 11.7 Count sequence and required decoding of a 4-bit twisted-ring counter.

Sequence number	Flip-flop outputs $Q3Q2Q1Q0$	AND gate required for output
1	0000	$Q3'Q0'$
2	1000	$Q3Q2'$
3	1100	$Q2Q1'$
4	1110	$Q1Q0'$
5	1111	$Q3Q0$
6	0111	$Q3'Q2$
7	0011	$Q2'Q1$
8	0001	$Q1'Q0$

11.15 Available MSI Counter ICs

The design of asynchronous and synchronous counters using flip-flops has been discussed in Sections 11.11 and 11.12 respectively. Some counters are available as MSI ICs and are given in Table 11.8.

IC 7493 is introduced here. IC 7493 is a 4-bit binary asynchronous up counter. The pin assignment of the IC is shown in Figure 11.53. The IC has two separate up counters, one ÷2 counter and one ÷8 counter. CK_0 is the clock input and Q_0 is the counter output of the ÷2 counter. CK_1 is the clock input and $Q_3Q_2Q_1$ are the counter outputs of the ÷8 counter. The master reset inputs MR_1 and MR_2 internally connected by a NAND gate to provide the active-LOW clear input to all flip-flops. To make the counter a ÷16 counter, Q_0 is to be externally connected to CK_1.

Table 11.8 Available MSI counter ICs.

IC No.	Description	Features
7490, 74290	BCD synchronous counter	Set, reset
7492	Divide-by-12 synchronous counter	Reset
7493, 74293	4-bit binary asynchronous (ripple) counter	Reset
74197	Presettable 4-bit binary asynchronous (ripple) counter	Reset, load
74390	Dual decade synchronous counters	Reset
74393	Dual 4-bit binary asynchronous (ripple) counters	Reset
74490	Dual BCD asynchronous (ripple) counters	Set, reset

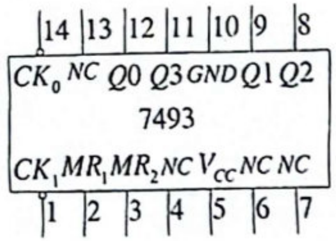

14
13
12
11
10
9
8
CK_0 NC Q0 Q3 GND Q1 Q2
7493
CK_1 MR_1 MR_2 NC V_{CC} NC NC
1
2
3
4
5
6
7

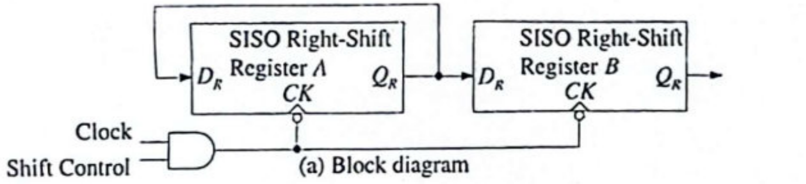
Figure 11.53 IC 7493 4-bit binary asynchronous counter.

11.16 Applications of Counters

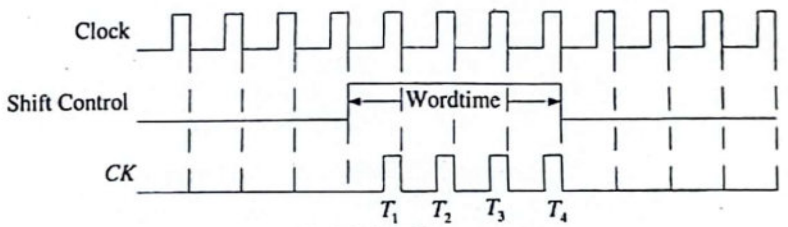
The sequence of operations in a digital system is specified by a control unit. The control unit that supervises the operations in a digital system would normally consist of timing signals that determine the time sequence in which the operations are executed. The timing sequences in the control unit can be easily generated by means of counters. Applications of counters in timing sequences generation as well as other popular applications of counters are discussed in this section.

11.16.1 Word-Time generator

A digital system is said to operate in a serial mode when information is transformed and manipulated one bit at a time. The content of one register is transferred to another by shifting the bits from one register to the other. The information is transferred one bit at a time by shifting the bits out of the source register into the destination register. The serial transfer of information from register *A* to register *B* is done with SISO right-shift registers, as shown in the block diagram of Figure 11.54(a). The serial output (Q_R) of register *A* goes to the serial input (D_R) of register *B*. To prevent the loss of information stored in the source register, the *A* register is made to circulate its information by connecting the serial output to its serial input terminal. The shift-control input determines when and by how many times the registers are shifted. This is done by the AND gate that allows clock pulses to pass into the *CK* terminals only when the shift-control is 1. Suppose the shift registers have four bits each. The control unit that supervise the transfer must be designed in such a way that it enables the shift registers, through the shift-control signal, for a fixed time duration (known as *word-time*) equal to four clock pulses. This is shown in the timing diagram of Figure 11.54(b). The shift-control signal is synchronized with the clock and changes value just after the negative edge of a clock pulse. The next four clock pulses find the shift-control signal in the 1 state, so the output of the AND gate connected to the *CK* terminals produces the four pulses T_1, T_2, T_3, and T_4. The fourth pulse changes the shift-control to 0 and the shift registers are disabled.



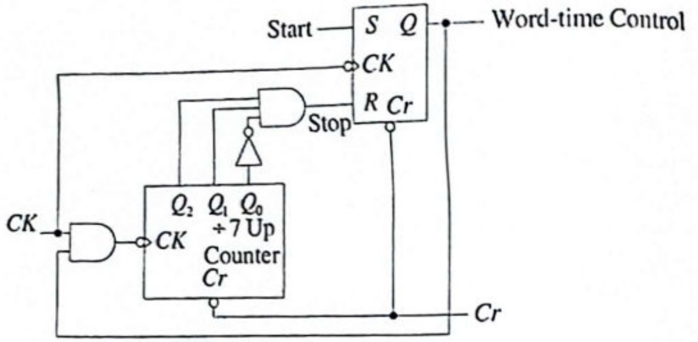
(a) Block diagram



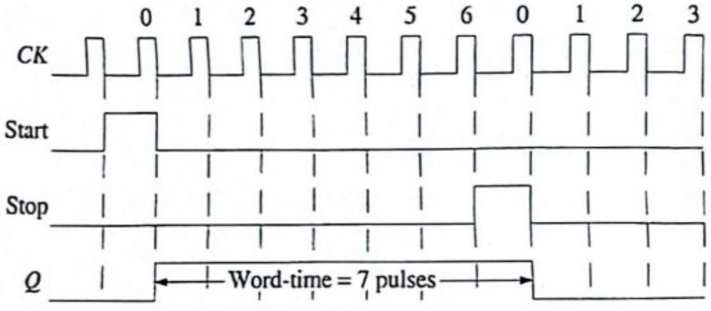
(b) Timing diagram

Figure 11.54 Serial transfer from register *A* to register *B*.

The control unit in a serial digital system must generate a *word-time* signal that stays ON for a number of pulses equal to the number of bits in the shift registers. The word-time signal can be generated by means of a counter that counts the required number of pulses. Assume that the word-time signal to be generated must stay ON for a period of seven clock pulses. Figure 11.55(a) shows a logic diagram that accomplishes this task. Initially, the ÷7 up counter is cleared to 000 and the S-R flip-flop output *Q* is reset to 0. A start signal will set the flip-flop output *Q*. The output of this flip-flop supplies the word-time control and also enables the counter. After the count of seven pulses, the flip-flop is reset and *Q* goes to 0. The timing diagram of Fig 11.55(b) demonstrates the operation of the circuit. The start signal is synchronized with the clock and stays ON for one clock pulse period. After *Q* is set to 1, the counter starts counting the clock pulses. When the counter reaches the count 110 (decimal 6), it sends a stop signal to the reset input of the flip-flop. The stop signal becomes a 1 after the negative-edge transition of pulse 6. The next clock pulse switches the counter to the 000 state and also clears *Q*. Now the counter is disabled and the word-time signal stays at 0. Note that the word-time control stays ON for a period of seven pulses.


Start
S Q
Word-time Control
CK
R Cr
Stop
CK
Q_2 Q_1 Q_0
CK ÷7 Up Counter
Cr
Cr

(a) Logic diagram


0 1 2 3 4 5 6 0 1 2 3
CK
Start
Stop
Q
Word-time = 7 pulses

(b) Timing diagram

Figure 11.55 Generation of word-time control for serial operations.

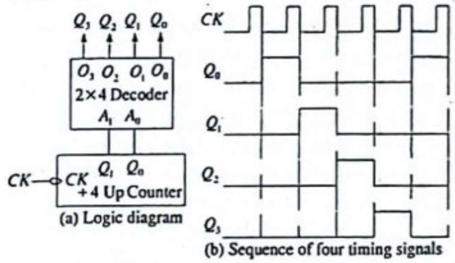
11. 16.2 Timing Signals Generation

In a parallel mode of operation, a single clock pulse can specify the time at which an operation should be executed. The control unit in a digital system that operates in a parallel mode must generate timing signals that stay ON for only one clock pulse period, but these timing signals must be distinguished from each other.

Timing signals that control the sequence of operations in a digital system can be generated with a ring counter. In a ring counter, only one flip-flop is set at any particular time and all others are cleared. The single bit is shifted from one flip-flop to the other to produce the sequence of timing signals.

The timing signals can also be generated by a $\div 2^n$ up counter along with a n-to-2^n-line decoder. A logic diagram for generating four timing signals is shown in Figure 11.56(a). The decoder decodes the four states of the counter and generates the required sequence of timing signals as shown in Figure 11.56(b).

The timing signals, when enabled by the clock pulses, will provide multiple-phase clock pulses. For example, if Q_0 is ANDed with CK, the output of the AND gate will generate clock pulses at one-fourth the frequency of the master-clock pulses. Multiple-phase clock pulses can be used for controlling different registers with different time scales.

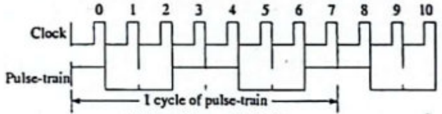

Q_3 Q_2 Q_1 Q_0
O_3 O_2 O_1 O_0
2×4 Decoder
A_1 A_0
CK
CK Q_1 Q_0
÷4 Up Counter
CK
Q_0
Q_1
Q_2
Q_3

(a) Logic diagram (b) Sequence of four timing signals

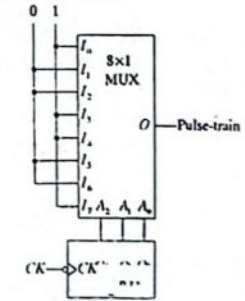
Figure 11. 56 Generation of timing signals.

11.16.3 Pulse-Train Generator

An arbitrary pulse-train with a period of 2^n clock pulse duration can be generated using a $\div 2^n$ up counter along with a $2^n \times 1$ multiplexer. An arbitrary pulse-train with a period of 8 clock pulse duration is shown in Figure 11.57(a). The logic diagram of Figure 11.57(b) generates this pulse-train. The $Q_2Q_1Q_0$ outputs of the ÷8 up counter are connected to the $A_2A_1A_0$ address lines of the 8×1 multiplexer. The inputs of the multiplexer is connected to 0 or 1 depending on the pulse-train. For example, at clock no 0, the value of the pulse-train is 1 and the I_0 input of the multiplexer is connected to 1. At clock no 1, the value of the pulse-train is 0 and the I_1 input of the multiplexer is connected to 0. As clock pulses are applied to the ÷8 up counter, it counts from 0 to 7 and repeats the sequence. Depending on the count value, the corresponding input of the multiplexer is connected to the output of the multiplexer and the desired pulse-train is generated.


0 1 2 3 4 5 6 7 8 9 10
Clock
Pulse-train
1 cycle of pulse-train

(a) An arbitrary pulse-train


0 1
I_0
I_1
I_2
I_3
I_4
I_5
I_6
I_7
8×1 MUX
O
Pulse-train
A_2 A_1 A_0
CK
CK

(b) Logic diagram

Figure 11.57 Generation of pulse-train.

11.16.4 Digital Clock

The logic diagram of a simple digital clock is shown in Figure 11.58. The 7-segment displays are divided into three groups - hour, minute, and second. The rightmost ÷60 up counter counts second, the middle ÷60 up counter counts minute, and the leftmost ÷20 up counter counts hour. All the counters can be reset by applying a 0 to the *Cr* input. A clock of 1 Hz (clock per second) is applied to the *CK* input of the second-counter. The second-counter changes its count in every second and counts from 00 to 59. The MSB of the second-counter is connected to the *CK* input of the minute-counter. When the second-counter goes from 59 to 00, the *CK* input of the minute-counter encounters a negative transition and changes its count. The minute-counter counts from 00 to 59. The MSB of the minute-counter is connected to the *CK* input of the hour-counter. When the minute-counter goes from 59 to 00, the *CK* input of the hour-counter encounters a negative transition and changes its count. When the count of the hour-counter reaches to 13, the hour-counter is reset to 00. Therefore, the hour-counter counts from 00 to 12.

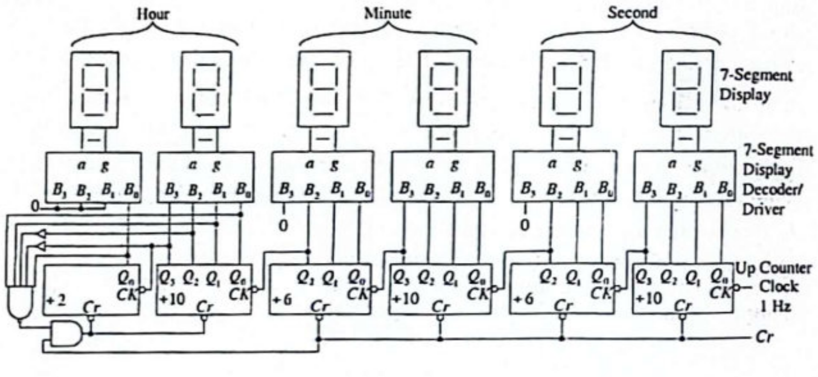


Figure 11.58 Block diagram of a digital clock.

Exercises

11.1 The content of a 4-bit SISO register is initially 1101. The register is shifted six times to the right, with the serial input being 101101. What is the content of the register after each shift?

11.2 The content of a 4-bit SISO register is initially 1001. The register is shifted five times to the left, with the serial input being 01101 (the left most bit is the LSB). What is the content of the register after each shift?

11.3 The content of a 4-bit SIPO register is initially 0000. The register is shifted six times to the right, with the serial input being 110011. What is the parallel output after the third shift?

11.4 The content of a 4-bit SIPO register is initially 1111. The register is shifted five times to the left, with the serial input being 10101 (the left most bit is the LSB). What is the parallel output after the fourth shift?

11.5 Design a 4-bit PIPO register with asynchronous loading using S-R flip-flops.

11.6 Design a 4-bit PISO right-shift register with synchronous loading using J-K flip-flops.

11.7 Design a 4-bit PISO right-shift register with asynchronous loading using J-K flip-flops.

11.8 Design a 4-bit PISO left-shift register with asynchronous loading using T flip-flops.

11.9 Design a 4-bit PISO left-shift register with asynchronous loading using S-R flip-flops.

11.10 Design a 4-bit PISO bi-directional register with synchronous loading using D flip-flops.

11.11 Design a 4-bit universal register with synchronous loading using D flip-flops.

11.12 A SISO register is often used to delay a digital signal by an integral number of clock cycles. If the waveforms of Figure 11.59 are applied to the input of a 10-bit SISO register, how long will it take the *D* waveform to appear at the serial output of the register?

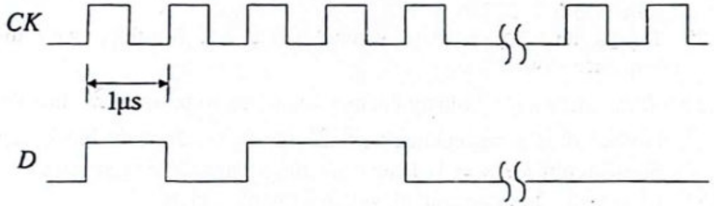


Figure 11.59 Waveforms for Exercise 11.12.

11.13 Design a ÷16 up/down ripple counter using T flip-flops.

11.14 Design a ÷16 up/down asynchronous counter with provision for asynchronous loading using J-K flip-flops.

11.15 Design a ÷5 up counter using T flip-flops.

11.16 Design a ÷64 up ripple counter using ÷8 up ripple counters.

11.17 Design a ÷64 down ripple counter using ÷8 down ripple counters.

11.18 Design a ÷128 up asynchronous counter using ÷16 and ÷8 up asynchronous counters.

11.19 Design a mod-12 up counter using a mod-16 up presettable asynchronous counter.

11.20 Design a mod-13 down counter using a mod-16 down presettable asynchronous counter.

11.21 Draw the logic diagram of a 4-bit binary up ripple counter using T flip-flops that trigger on the positive edge.

11.22 Draw the logic diagram of a 4-bit binary down ripple counter using T flip-flops that trigger on the positive edge.

11.23 How many flip-flops must be complemented in a 10-bit binary ripple up counter to reach the next count after 0111111111?

11.24 How many flip-flops must be complemented in a 10-bit binary ripple down counter to reach the next count after 0111111111?

11.25 Determine the count sequence of the ripple counter shown in Figure 11.60.

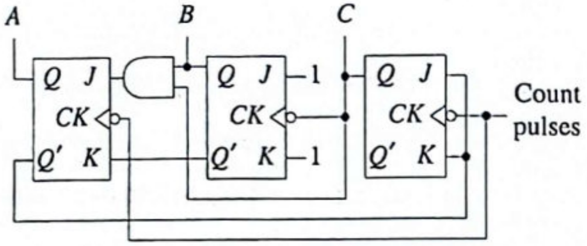


Figure 11.60 Ripple counter for Exercise 1.25.

11.26 Design a decade synchronous counter to count in the Excess-3 code sequence using J-K flip-flops.

11.27 Design a synchronous BCD counter using J-K flip-flops.

11.28 Design an odd sequence counter using S-R flip-flops with the count sequence 0, 1, 3, 2, 6, 4, 5, 7.

11.29 Design an odd sequence counter using J-K flip-flops with the count sequence 0, 4, 2, 1, 6.

11.30 Show that an n-bit binary counter connected to an n-to-2^n line decoder is equivalent to a ring counter with 2^n flip-flops. Draw the block diagrams of both circuits for $n = 3$. How many timing signals are generated?

11.31 Construct a Johnson counter with ten timing signals.

11.32 Construct a word-time control that stays ON for a period of 16 clock pulses.

11.33 Construct a word-time control that stays ON for a period of 13 clock pulses.

11.34 Construct a timing signals generator with 10 timing signals.

11.35 Construct a timing signals generator with 16 timing signals.

11.36 Design pulse-train generators to generate the pulse-trains of Figure 11.61.

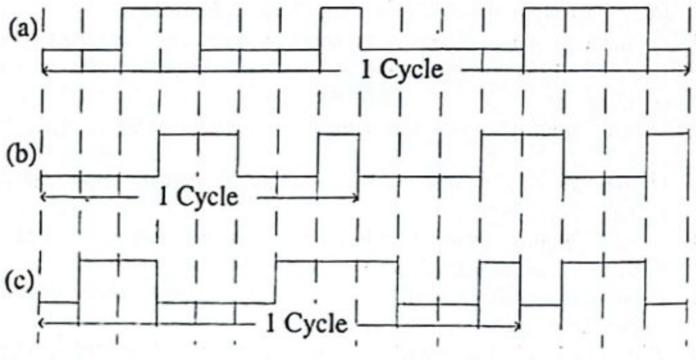


Figure 11.61 Arbitrary pulse-trains for Problem 11.36.

11.37 Design a digital watch that displays time of day in *HH:MM:SS* format with *HH* ranging from 00 to 23, *MM* ranging from 00 to 59, and *SS* ranging from 00 to 59.

Further Reading

1. *The TTL Data Book.* Dallas, Texas Instruments, 1988.
2. R. P. Jain, *Modern Digital Electronics*, 2nd ed., Tata McGraw-Hill Publishing Company Limited, India, 1998.
3. *Microprocessor Data Hand Book*, Revised Ed. BPB Publications, India, 2000.
4. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
5. D. C. Green, *Digital Electronics*, 5th ed., Pearson Education, 2001.
6. Ronald J. Tocci and Neal S. Widmer, *Digital Systems: Principles and Applications*, 8th ed., Prentice-Hall, Inc., 2001.
7. M. Morris Mano, *Digital Design*, 3rd ed., Pearson Education, 2002.
8. Samuel C. Lee, *Digital Circuits and Logic Design*, Prentice-Hall, Inc., 2002.

CHAPTER 12
MEMORIES AND PROGRAMMABLE LOGIC DEVICES

12.1 Introduction

In the previous chapters, design procedures for combinational and sequential circuits have been discussed. Various ICs for performing basic digital operations and other functions, such as adders, comparators, encoders, decoders, code converters, multiplexers, demultiplexers, registers, and counters, etc. have also been discussed. A digital system, however, invariably requires a facility for storing digital information. The subsystem of a digital system, which provides the storage facility, is referred to as the *memory*. Memory units are large-scale integrated (LSI) devices, which provide the storage facility in a digital system. In this chapter, fundamental concepts of two types of memory units – random-access memory (RAM) and read-only memory (ROM) are introduced.

Various ICs for performing basic digital operations and other functions discussed in the previous chapters are referred to as *fixed-function ICs*, i.e., each one of them performs a specific, fixed function. These devices are designed by their manufacturers and are manufactured in large quantities to meet the needs of a wide variety of applications and are readily available. To design a circuit, a designer can select the available ICs most appropriate for the circuit. The design may have to be modified to meet the special requirement of these devices. Another approach of designing a digital circuit is the use of *programmable logic devices (PLDs)*. A PLD is a large-scale integrated (LSI) circuit that is user configurable and is capable of implementing logic functions. PLDs can be programmed to incorporate complex logic functions within an LSI circuit. In this chapter, fundamental concepts of various PLDs and their uses are also discussed.

12.2 Memories and Their Classifications

A memory unit is a collection of storage cells together with associated circuits needed to transfer information in and out of storage. Memory devices can be classified on the basis of their mode of access as (i) *random-access memory (RAM)* and (ii) *read-only memory (ROM)*.

A *random-access memory (RAM)* can transfer the stored information out (read) and is also capable of receiving new information in for storage (write). A more appropriate name for such a memory would be *read-write memory*. In RAMs, storage cells can be accessed for information transfer to or from any desired random location and hence the name.

A *read-only memory (ROM)* is a memory device used to store information, which is permanent in nature. It can only transfer the stored information out (read). The process of writing information into this memory is done outside the system where it is used. The process of entering information into a ROM is referred to as programming the ROM. Depending on the programming process employed the ROMs are classified as:

1. *Mask Programmable Read-Only Memory* (referred to as *ROM*). In ROMs, the data pattern must be programmed as part of the fabrication process. Once programmed, the data pattern can never be changed.
2. *Programmable Read-Only Memory* (referred to as *PROM*). A PROM is electrically programmable, that is, the data pattern is defined after final packaging rather than when the device is fabricated. The programming is done with a device referred to as PROM programmer.
3. *Erasable Programmable Read-Only Memory* (referred to as *EPROM*). In EPROMS, data can be written any number of times, that is, they are reprogrammable.

12.3 Random-Access Memory

12.3.1 Organization of Random-Access Memory

A memory unit stores binary information in groups of bits called *words*. A word in memory is an entity of bits that move in and out of storage as a unit. Most memories use words that are multiples of bytes in length. The capacity of a memory unit is usually stated as the total number of bytes that it can store. A memory unit is characterized by the number of words it contains and the number of bits in each word. For example, a 1024×16 RAM contains 1024 words of memory and length of each word is 16 bits. Therefore, the capacity of the RAM is 2048 bytes.

The communication between a memory and its environment is achieved through data input and output lines, address lines, and control lines that specify the direction of transfer. The block diagram of a 8×4 RAM unit is shown in Figure 12.1. The 4 data input lines (I_0 through I_3) provide the information to be stored in memory and the 4 data output lines (O_0 through O_3) supply the information coming out of memory. The 3 address lines (A_0 through A_2) specify the particular word chosen among the 8 available. Each word in a memory is assigned an identification number, called an *address*, starting from 0 and continuing with 1, 2, 3, up to $2^k - 1$, where k is the number of address lines. The read/write (*R/W*) control line specify the direction of transfer desired: a 0 input causes data to be transferred into the memory (write) and a 1 causes the data to be transferred out of memory (read).

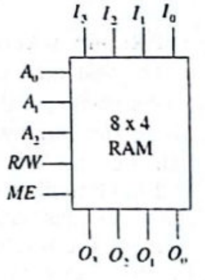


Figure 12.1 Block diagram of an 8×4 RAM unit.

Write and Read Operations

The two operations that a RAM can perform are the write and read operations. The write signal specifies a transfer-in operation and the read signal specifies a transfer-out operation. On accepting one of these control signals, the internal circuits inside the memory provide the desired function. The steps that must be taken for the purpose of transferring a new word to be stored into RAM are as Follows:

1. Transfer the address of the desired word to the address lines.
2. Transfer the data bits that must be stored in memory to the data input lines.
3. Set the *R/W* input to 0.

The RAM unit will then take the bits from the input data lines and store them in the word specified by the address lines.

The steps that must be taken for the purpose of transferring a stored word out of RAM are as follows:

1. Transfer the address of the desired word to the address lines.
2. Set the *R/W* input to 1.

The RAM unit will then take the bits from the word that has been selected by the address and apply them to the output data lines. The content of the selected word does not change after reading.

The memory enable (*ME*), sometimes called the chip select (*CS*), is used to enable the particular memory chip in a multi-chip implementation of a large memory. When the *ME* input is inactive, the memory chip is not selected and no operation is performed. When the *ME* input is active, *R/W* input determines the operation to be performed.

The memory operations that result from the control inputs are summarized in Table 12.1.

Table 12.1 Control inputs to RAM chips.

Memory Enable (*ME*)	Read/Write (*R/W*)	Memory Operation
0	x	None
1	0	Write to selected word
1	1	Read from selected word

The stored data in a RAM is lost when the power is put off. Hence the RAM is called *volatile memory*.

12.3.2 Internal Construction of Random-Access Memory

The internal construction of a $m \times n$ random-access memory of m words with n bits per word consists of $m \times n$ 1-bit memory cell (*MC*) and associated decoding circuits for selecting individual words. The memory cell is the basic building block of a memory unit. The memory cell is discussed in Section 9.10 (see Figure 9.28), which is shown here in Figure 12.2 in block diagram form. A memory cell stores one bit in its internal flip-flop. It has three inputs and one output. The address input enables the cell for reading or writing. The read/write input determines the cell operation when it is selected. A 1 in the read/write input provides the read operation and a 0 provides the write operation.

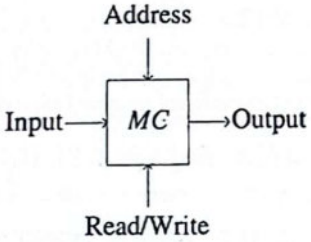


Figure 12.2 Block diagram of a memory cell.

The logical construction of a 4x4 RAM is shown in Figure 12.3. It consists of 4 words of 4 bits each and has a total of 16 memory cells. Each block labeled *MC* represents the memory cell with its three inputs and one output as specified in Figure12.2. A memory with four words need two address lines. The two address inputs go through a 2x4 decoder with active-HIGH output to select one of the four words. The active-HIGH enable input of the decoder is used as the memory enable (*ME*) input. When the *ME* input is 0, all outputs of the decoder are 0 and none of the memory words are selected. With the *ME* input is 1, one of the four words is selected, dictated by the value in the two address lines. Once a word has been selected, the read/write (*R/W*) input determines the operation. During the read operation, the four bits of the selected word go through OR gates to the output terminals. During the write operation, the data available in the input lines are transferred into the four memory cells of the selected word. The memory cells that are not selected are disabled and their previous values remains unchanged. When the *ME* input is 0, none of the words are selected and the contents of all cells remain unchanged regardless of the value of the read/write (*R/W*) input.

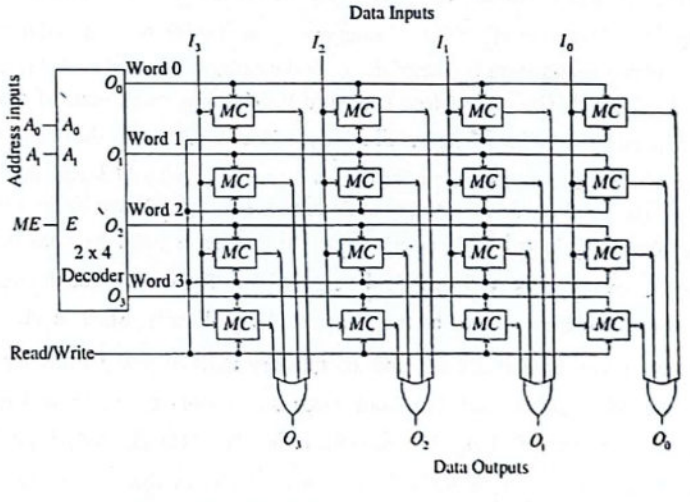


Figure 12.3 Logical construction of 4x4 RAM.

12.3.3 Expansion of RAM Size

In many memory applications, the required RAM capacity, i.e., the number of words and/or word size, cannot be satisfied by a single available RAM IC chip. Therefore, several similar chips have to be combined suitably to provide the desired number of words and/or word size.

Construction of a 16×4 RAM using eight 4×2 RAMs is shown in Figure 12.4. Address lines A_1 and A_0 of all eight memory units are connected together to form the address lines A_1 and A_0 of the combined memory system. The address lines A_3 and A_2 of the combined memory system are connected to the two addresss lines of the 2×4 decoder. The output O_0 of the decoder is connected to the *ME* inputs of the memory units of row 0 (memory units $M1$ and $M0$). Similarly, the output O_1 of the decoder is connected to the *ME* inputs of the memory units of row 1 (memory units $M3$ and $M2$). The output O_2 of the decoder is connected to the *ME* inputs of the memory units of row 2 (memory units $M5$ and $M4$). And the output O_3 of the decoder is connected to the *ME* inputs of the memory units of row 3 (memory units $M7$ and $M6$). Read/write (*R/W*) inputs of all eight memory units are connected together to form the read/write (*RW*) input of the combined memory system. The corresponding outputs of the memory units of each column are ORed to form the output of the combined memory system. Now the word size becomes 4 and the number of words becomes 16.

The most significant two address lines A_3A_2 select one of the four rows and the least significant two address lines A_1A_0 select one of the four words of the selected row. When $A_3A_2 = 00$, the output O_0 of the decoder is 1 and the other outputs of the decoder are 0. Therefore, only the memory units of row 0 (memory units $M1$ and $M0$) are enabled. The word inside the memory units of row 0 is then selected by the value of A_1A_0. Similarly, when $A_3A_2 = 01$, the output O_1 of the decoder is 1 and the memory units of row 1 (memory units $M3$ and $M2$) are enabled. The word inside the memory units of row 1 is then selected by the value of A_1A_0. When $A_3A_2 = 10$, the output O_2 of the decoder is 1 and the memory units of row 2 (memory units $M5$ and $M4$) are enabled. The word inside the memory units of row 2 is then selected by the value of A_1A_0. Finally, when $A_3A_2 = 11$, the output O_3 of the decoder is 1 and the memory units of row 3 (memory units $M7$ and $M6$) are enabled. The word inside the memory units of row 3 is then selected by the value of A_1A_0. For example, if the address is $A_3A_2A_1A_0 = 1011$ (decimal 11), then $A_3A_2 = 10$ and the memory units of row 2 (memory units $M5$ and $M4$) are enabled. As $A_1A_0 = 11$, word 3 of row 2 is selected, which is word 11 of the combined memory system.

The active-HIGH enable input of the 2×4 decoder is used as the memory enable (*ME*) input of the combined memory system. When $ME = 0$, all outputs of the decoder are 0 and consequently all memory units are disabled making the combined memory system disabled.

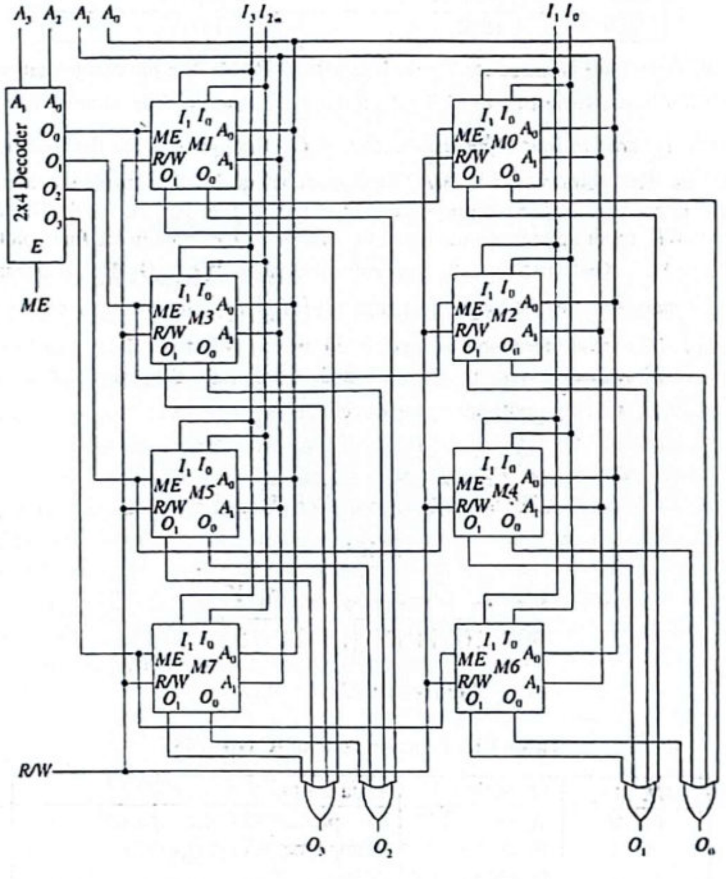

A_3 A_2 A_1 A_0
I_3 I_2
I_1 I_0
2x4 Decoder
A_1 A_0
O_0
O_1
O_2
O_3
E
ME
ME M1 R/W I_1 I_0 A_0 A_1 O_1 O_0
ME M0 R/W I_1 I_0 A_0 A_1 O_1 O_0
ME M3 R/W I_1 I_0 A_0 A_1 O_1 O_0
ME M2 R/W I_1 I_0 A_0 A_1 O_1 O_0
ME M5 R/W I_1 I_0 A_0 A_1 O_1 O_0
ME M4 R/W I_1 I_0 A_0 A_1 O_1 O_0
ME M7 R/W I_1 I_0 A_0 A_1 O_1 O_0
ME M6 R/W I_1 I_0 A_0 A_1 O_1 O_0
R/W
O_3
O_2
O_1
O_0

Figure 12.4 16×4 RAM using 4×2 RAMs.

12.3.4 Available RAM ICs

Some of the commonly used RAM ICs are given in Table 12.2.

Table 12.2 Available RAM ICs.

IC No	RAM Organization	Technology
1101A	256×1	MOS
2101A	256×4	MOS
2114	1024×4	MOS
5101	256×4	CMOS
6116	2048×8	CMOS
7489	16×4	TTL

IC type 7489 is introduced here. It is a 16×4 RAM. The pin assignment of IC type 7489 is shown in Figure 12.5. A_3 through A_0 are the four address lines. I_3 through I_0 are the four input data lines and O_3' through O_0' are the four output data lines. The memory enable (ME') input must be equal to 0 to enable the memory. If ME' is equal to 1, the memory is disabled and all four outputs are 1. The write enable (WE) input determines the type of operation as indicated in the function table of Table 12.3. The write operation is performed when $WE = 0$. This is a transfer of the binary number from the data inputs ($I_3I_2I_1I_0$) into the selected word in memory. The read operation is performed when $WE = 1$. This transfers the complement value stored in the selected word into the output data lines ($O_3'O_2'O_1'O_0'$). The outputs are open-collector.

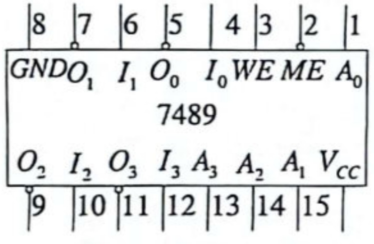


Figure12.5 IC type 7489.

Table 12.3 Function table of IC type 7489.

ME′	WE	Operation	Data Output
0	0	Write	Complements of data inputs
0	1	Read	Complements of data inputs
1	x	Disable	All 1s

12.4 Read-Only Memory

12.4.1 Organization of Read-Only Memory

A read-only memory (ROM) is essentially a storage device in which permanent data is stored. The data must be specified by the designer and is then embedded in the unit to form the data pattern. Once the data pattern is established, it stays within the unit even when the power is turned off and on again. Similar to a RAM, a ROM is also characterized by the number of words and the number of bits per word.

The block diagram of a 16×4 ROM is shown in Figure 12.6. It consists of 4 address lines (A_0 through A_3) and 4 output lines (D_0 through D_3). The particular word selected that is presently available on the output lines is determined by the address inputs.

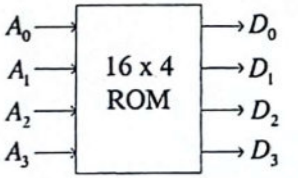


Figure 12.6 Block diagram of a 16×4 ROM.

12.4.2 Internal Construction of Read-Only Memory

The logical construction of a 4x4 ROM is shown in Figure 12.7. The two address inputs are decoded into 4 lines by means by a 2x4 decoder with active-HIGH output. The 4 outputs of the decoder are connected through electronic fuses to 4 OR gates for four outputs. The ROM is programmed by blowing the fuses corresponding to the 0 bits of the data patterns. Figure 12.8 shows the logical construction of a 4x4 ROM after programming for storing the data pattern shown in the figure. It is assumed here that an open input to an OR gate behaves as a 0 input. The active-HIGH enable input (E) of the decoder is used as the memory enable (ME) input of the ROM unit. When the decoder is disabled, all the outputs of the decoder are 0 and consequently all the outputs of the ROM are 0. It should be noted that the construction of a ROM unit given here is logical construction. The actual physical construction of a ROM unit is beyond the scope of this book.

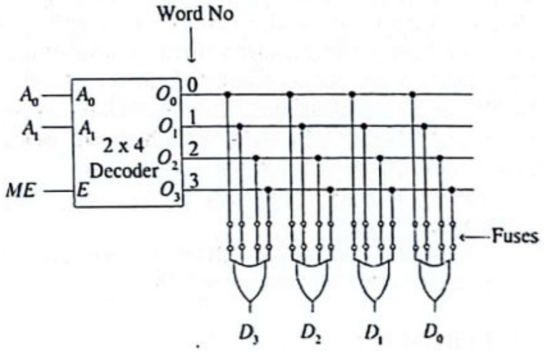


Figure 12.7 Logical construction of a 4x4 ROM.

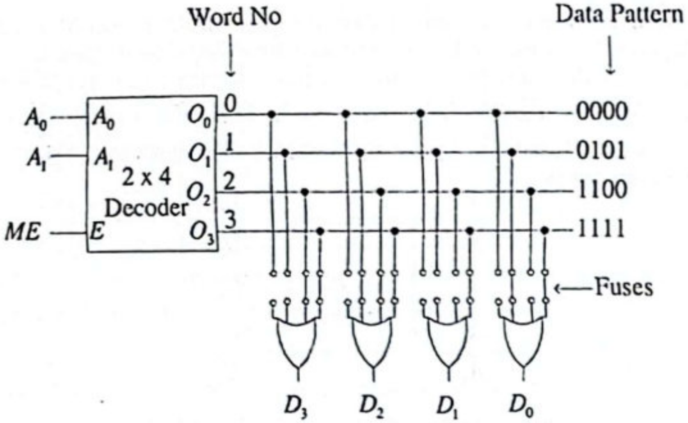


Figure 12.8 Logical construction of a 4×4 ROM after programming for a data patter.

Programming the ROMs

The ROM may be programmed in two different ways. The first is called *mask programming* and is done by the manufacturer during the fabrication process of the unit. The second way is to program a programmable read-only memory (PROM) using a special device called *PROM programmer*. PROM units originally contains all 0s (or all 1s) in every bit of the stored words. The fuses in the PROM are blown by the PROM programmer. A blown fuse defines one binary state and an unbroken link represents the other state. This allows the user to program the unit in the laboratory to achieve the desired data pattern. It should be noted that, in any cases, all procedures for programming ROMs are hardware procedures.

The hardware procedure for programming ROMs or PROMs is irreversible and, once programmed, the fixed pattern is permanent and cannot be altered. A third type of unit available is called *erasable PROM (EPROM)*. EPROMs can be restructured to the initial value (all 0s or all 1s) even though they have been changed previously. When an EPROM is placed under a special ultraviolet light for a given period of time, the EPROM restores its original all 0s (or all 1s) pattern. After erasure, the EPROM can be reprogrammed. Certain PROMs can be erased with electrical signals instead of ultraviolet light, and these are called *electrically erasable PROMs (EEPROMs)*.

Expansion of ROM size

The number of word and/or the word size of a ROM system can be extended using the similar procedure followed for the extension of RAM size.

12.4.3 Available EPROM ICs

Some of the commonly used EPROM ICs are given in Table 12.4.

Table 12.4 Some available EPROM ICs.

IC No	Organization
2708	1K×8
2716	2K×8
2732	4K×8
2764	8K×8
2816	2K×8
27128	16K×8
27256	32K×8
27512	64K×8

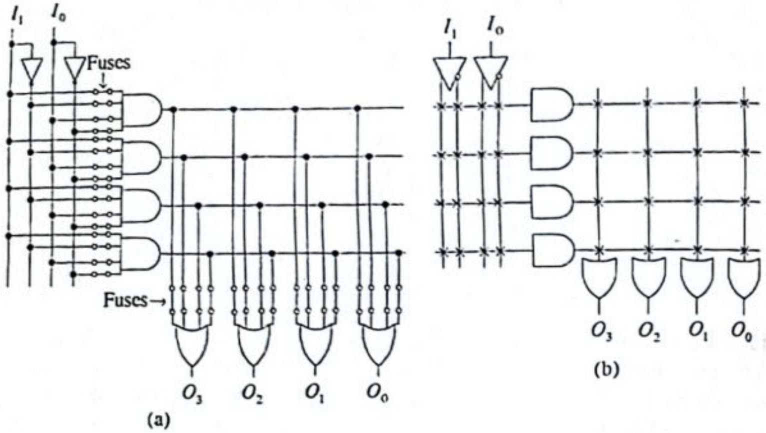
12.5 Programmable Logic Devices

A programmable logic device (PLD) is an LSI chip that contains a regular structure and allows the designer to customize it for any specific application, i.e., it is programmed by the user to perform a function required for his application. The PLDs allow designer more flexibilities to experiment with designs because these can be reprogrammed.

Because of various advantages of PLDs, a large number of PLDs have been produced by IC manufacturers with variety of felexibilities and options available for a circuit designer and have become very popular.

A PLD is an IC with internal logic gates that are connected through electronic fuses. Programming the devices involves the blowing of fuses along the paths that must be disconnected so as to obtain a particular configuration. The gates in a PLD are divided into an AND array and an OR array that are connected together to provide an AND-OR sum of product implementation. The initial state of a PLD has all the fuses intact. Programming the devices involves the blowing of internal fuses to achieve a desired logic function.

General architecture of a PLD is shown in Figure 12.9(a). Each of the two input variables I_1 and I_0 have two outputs – normal and complemented. Each of the four inputs is connected to one of the four inputs of an AND gate through fuses. In this example architecture, four such AND gates are shown, which constitutes the AND array. Each of the four AND gates can be programmed to implement any product term of two variables. Each of the four outputs of four AND gates is connected to one of the four inputs of an OR gate through fuses. Four such OR gates are shown, which constitutes the OR array. The four OR gates realizes four functions O_3, O_2, O_1, and O_0. Each of the four OR gates can be programmed to implement any sum of product function of two variables.



(a)

(b)

Figure 12.9 General architecture of a PLD.

The example architecture of Figure 12.9(a) has only two input variables, four AND gates, and four OR gates. The circuit diagram is already quite cluttered. With many more inputs and gates the diagram would be more messy. For this reason, a simplified symbology is used to represent PLDs as in Figure 12.9(b). In Figure 12.9(b), input is represented by a buffer with two outputs - normal and complemented. A single line is shown going into the AND gate to represent the four inputs. The fuse connections from the inputs to the AND gate are indicated as ×. Similarly, a single line is shown going into the OR gate to represent the four inputs. The fuse connection from the output of the AND gates to the OR gate are also indicated as ×. In specific PLD architecture, some of the connections may be hard-wired, which is indicated as a dot (•). The absence of either a dot or a × indicates no connection.

Depending on the placement of fuses in the AND-OR array, there are three types of PLDs

1. *Programmable read-only memory (PROM).* It has a fixed (hard-wired) AND array and programmable fuses for the output OR gates. The PROM implements Boolean function in sum of minterms.
2. *Programmable array logic (PAL).* It has a fused programmable AND array and a fixed (hard-wired) OR array. The AND gates are programmed to provide the product terms for the Boolean functions that are logically summed in each OR gate.
3. *Programmable logic array (PLA).* It is the most flexible PLD, where both the AND and OR arrays can be programmed. The product terms in the AND array may be shared by any OR gate to provide the required sum of products implementation.

12.6 Programmable Read-Only Memory as Programmable Logic Device

12.6.1 Architecture of Programmable Read-Only Memory

Read-only memory (ROM) is introduced in Section 12.4. Logical construction of a 4×4 ROM is shown in Figure 12.7. It can also be seen from Section 8.3 that a 2×4 decoder with active-HIGH output produces four minters of two variables through four AND gates. Considering all these facts, the architecture of a 4×4 PROM is shown in Figure 12.10 (The memory enable input ME is not shown in this figure.). It consists of a fixed AND array and a fused programmable OR array. The four AND gates produce the four possible minterms of two variables. Each of the four minterms is connected to one of the four inputs of each OR gates through fuses. With all the fuses initially intact, each OR output will be 1. Each of the four outputs O_3, O_2, O_1, and O_0 can be programmed to implement any function of I_1 and I_0 by selectively blowing the appropriate fuses. PROMs are so designed that a blown OR input acts as a 0. For example, if we blew fuses at the intersections of the AND gates 0 and 3 with the OR gate 3, the O_3 output becomes

$$O_3 = 0 + I_1'I_0 + I_1I_0' + 0 = I_1'I_0 + I_1I_0'.$$

In a similar manner, we can program each of the four OR outputs to any desired function. Once all of the outputs have been programmed, the device will permanently generate the selected output function.

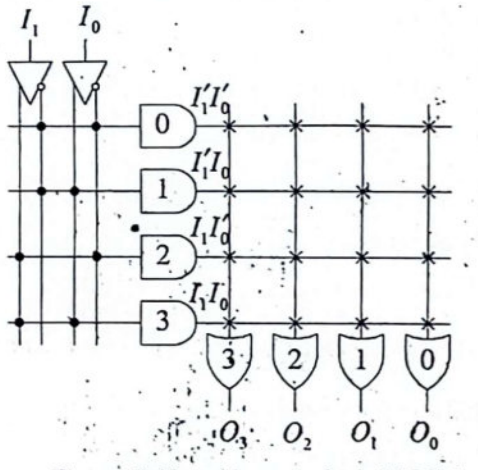


Figure 12.10 Architecture of a 4×4 PROM.

12.6.2 Boolean Function Implementation with Programmable Read-Only Memory

Any Boolean function can be expressed as sum of minterms. By blowing the fuses of those minterms not included in the function, each PROM output can be made to represent the Boolean function of one of the output variables in the combinational circuit. For an n-input, m-output combinational circuit, we need $2^n \times m$ PROM.

When a combinational circuit is implemented by means of a PROM, first the function is expressed by a truth table. Next, the minterms that specify an output of 1 are identified. These minterms are to be ORed to get the desired function. Thus, only those fuses that connect those minterms to the inputs of OR gate are to be left intact and others are to be blown. This procedure is illustrated for the combinational circuit specified by the following four functions in Figure 12.11.

$$F_0(x, y, z) = x + yz'$$

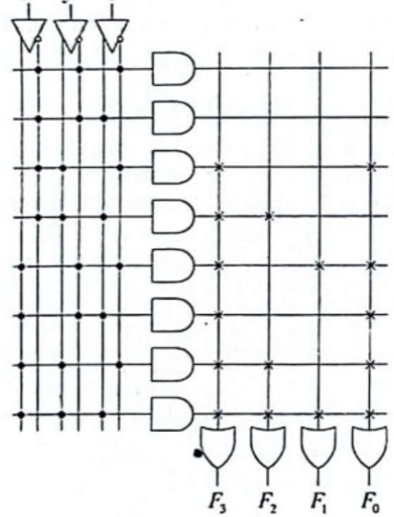
$$F_1(x, y, z) = xy'x' + xyz$$

$$F_2(x, y, z) = xy + yz$$

$$F_3(x, y, z) = x + y$$

x	y	z	F_3	F_2	F_1	F_0
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	1	0	0	1
0	1	1	1	1	0	0
1	0	0	1	0	1	1
1	0	1	1	0	0	1
1	1	0	1	1	0	1
1	1	1	1	1	1	1

(a) Truth table



(b) Fuse map

Figure 12.11 Implementation of functions using an 8×4 PROM.

12.7 Programmable Array Logic

12.7.1 Architecture of Programmable Array Logic

The PROM architecture is well suited for those applications where every possible input combination is required to generate the output functions. Examples are code converters. There are many applications that do not require that all input combinations be programmable. The logic functions shown in Figure 12.11, for example, do not use all of the available minterms. This has led to the development of a class of PLDs called *programmable array logic (PAL)*.

The architecture of a PAL differs slightly from that of a PROM. The PAL has the same AND and OR array as with PROMs, but in PAL the inputs to the AND gates are programmable while the inputs to the OR gates are hard-wired. This means that every AND gate can be programmed to generate any desired product of the input variables and their complements. Each OR gate is hard-wired to only some of the AND outputs. This limits each output function to a fixed number of product terms.

A PAL is characterized by the number of inputs, number of programmable AND gates, and number of fixed OR gates. The configuration of AND and OR arrays of a PAL with 4 inputs, 12 programmable AND gates, and 4 fixed OR gates is shown in Figure 12.12. Each AND gate has all 8 inputs (in normal and complemented form) with fuses intact, which can be programmed to generate 12 product terms. Each OR gate gets inputs from the outputs of only three AND gates shown by dot. Thus, there are four sections in the PAL unit of Figure 12.12, each being composed of three AND gates and one OR gate.

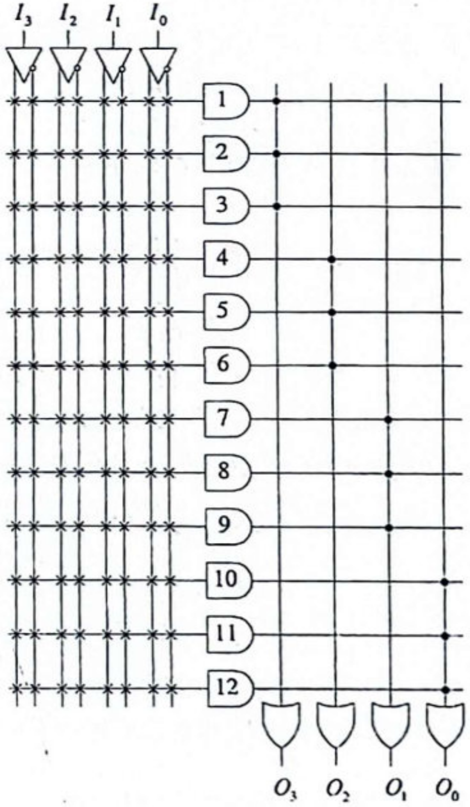


Figure 12.12 Typical PAL architecture.

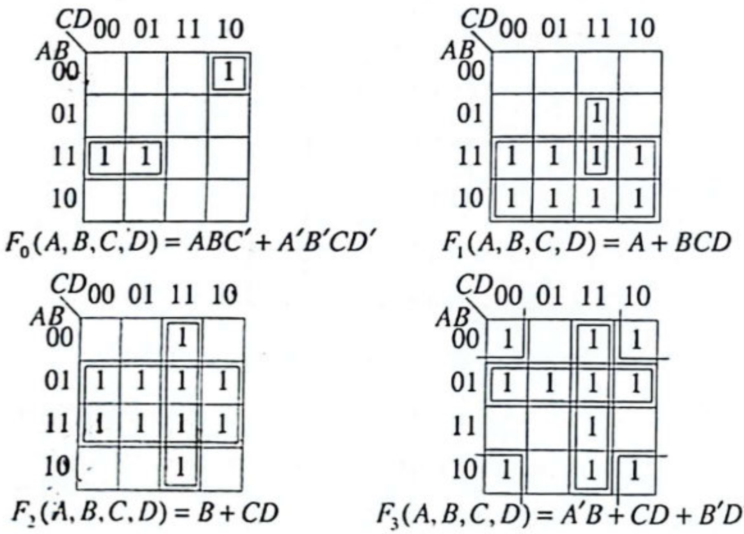
12.7.2 Boolean Function Implementation with Programmable Array Logic

When implementing with a PAL, the Boolean function must be simplified to fit into each section. As an example of using a PAL in the implementation of a combinational circuit, consider the following Boolean functions given in sum of minterms:

$$F_0(A,B,C,D)=\sum(2,12,13)$$
$$F_1(A,B,C,D)=\sum(7,8,9,10,11,12,13,14,15)$$
$$F_2(A,B,C,D)=\sum(3,4,5,6,7,11,12,13,14,15)$$
$$F_3(A,B,C,D)=\sum(0,2,3,4,5,6,7,8,10,11,15)$$

Simplification of the four functions to minimum number of terms is shown in Figure 12.13, which results in the following Boolean functions:

$$F_0(A,B,C,D)=ABC'+A'B'CD'$$
$$F_1(A,B,C,D)=A+BCD$$
$$F_2(A,B,C,D)=B+CD$$
$$F_3(A,B,C,D)=A'B+CD+B'D'$$



AB \ CD	00	01	11	10
00				1
01				
11	1	1		
10				

$F_0(A,B,C,D)=ABC'+A'B'CD'$

AB \ CD	00	01	11	10
00				
01			1	
11	1	1	1	1
10	1	1	1	1

$F_1(A,B,C,D)=A+BCD$

AB \ CD	00	01	11	10
00			1	
01	1	1	1	1
11	1	1	1	1
10			1	

$F_2(A,B,C,D)=B+CD$

AB \ CD	00	01	11	10
00	1		1	1
01	1	1	1	1
11			1	
10	1		1	1

$F_3(A,B,C,D)=A'B+CD+B'D'$

Figure 12.13 Simplification using K-map.

The PAL programming table for the four Boolean functions of the example is shown in Table 12.5. It consists of three columns. The first column lists the product terms numerically. The second column specifies the required paths between inputs and AND gates. The third column shows the function. The table is divided into four sections with three product terms in each to conform to the PAL of Figure 12.12. The first three sections need only two product terms to implement the Boolean functions. The last section needs three product terms. For each product term, the inputs are marked with 1, 0, or – (dash). If a variable in the product term appears in its normal form (uncomplemented), the corresponding input variable is marked with a 1. If it appears complemented (primed), the corresponding input variable is marked with a 0. If the variable is absent in the product term, it is marked with a dash. Each product term is associated with an AND gate.

Table 12.5 PAL program table.

Product term	AND inputs A B C D	Outputs
1	0 1 – –	$F_3=A'B+CD+B'D'$
2	– – 1 1	
3	– 0 – 0	
4	– 1 – –	$F_2=B+CD$
5	– – 1 1	
6	– – – –	
7	1– – –	$F_1=A+BCD$
8	– 1 1 1	
9	– – – –	
10	1 1 0 –	$F_0=ABC'+A'B'CD'$
11	0 0 1 0	
12	– – – –	

The fuse map for the PAL as specified in the programming table of Table 12.5 is shown in Figure 12.14. For each 1 or 0 in the table, we mark the corresponding intersection in the diagram with the symbol for an intact fuse. For each dash, we mark the diagram with blown fuses in both the normal and complement inputs. It is assumed that the open terminal in the AND gate behaves like a 1 input. If the AND gate is not used, we leave all its input fuses intact. Since the corresponding input receives both the normal and complemented value of each input variable, we have $AA'=0$ and the output of the AND gate is always 0.

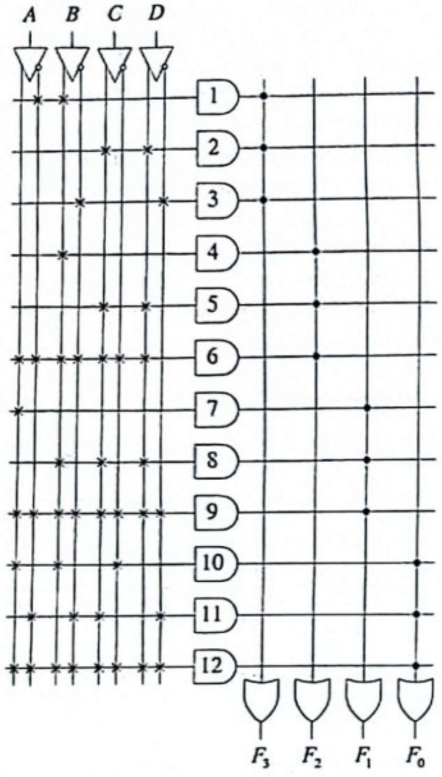


Figure 12.14 Fuse map for implementing functions of Table 12.5.

12.7.3 Available PAL Devices

Some of the available PAL devices are given in Table 12.6.

Table 12.6 Some available PAL devices.

Device	Inputs	Outputs
PAL 10L8	10	8
PAL 16L8	16	8
PAL 20L8	20	8
PAL 20L10	20	10
PAL 16R4	16	8
PAL 16R6	16	8

12.8 Programmable Logic Array

12.8.1 Architecture of Programmable Logic Array

A programmable logic array (PLA) combines the characteristics of the PROM and the PAL by providing both a programmable OR array and a programmable AND array. The number of AND and OR gates and their inputs are fixed for a given PLA. The AND gates provide the product terms, and the OR gates logically sum these product terms and thereby generate a sum of products expression. The size of the PLA is specified by the number of inputs, the number of product terms, and the number of outputs. It has M inputs, n product terms, and N outputs with $n < 2^M$, and can be used to implement a logic function of M variables with N outputs. Since all of the possible 2^M minterms are not available, logic minimization is required to accommodate a given logic function. The configuration of a 4 inputs, 12 product terms, and 4 outputs PLA is shown in Figure 12.15.

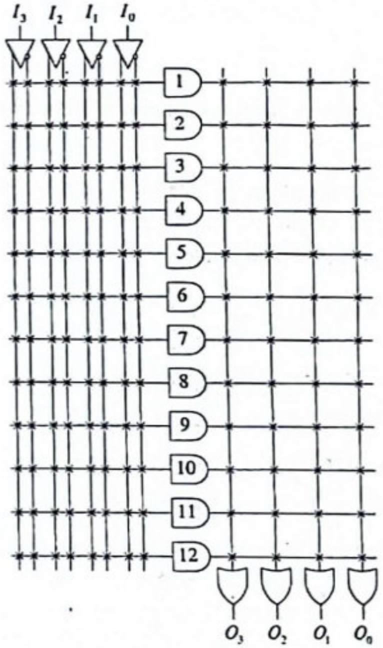


Figure 12.15 Architecture of a PLA.

In commercially available PLAs, the outputs of the OR-gates are made inverting/non-inverting. Typical output is shown in Figure 12.16. If the fuse of one of the inputs of EXOR gate is intact, then its effect is a logical 0 and the output is non-inverted (since $O \oplus 0 = O$). If the fuse is blown off, then its effect is logical 1 and the output is inverted (since $O \oplus 1 = O'$).

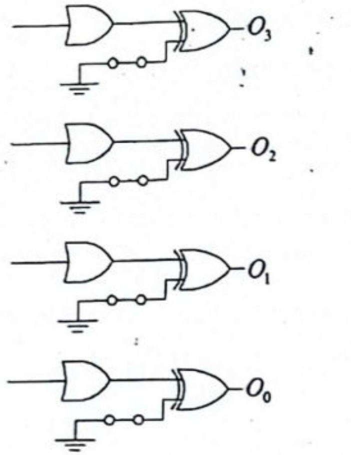


Figure 12.16 Inverting/non-inverting outputs of a PLA with 4 outputs.

The outputs are sometimes made tri-state buffered as shown in Figure 12.17. When the *CE* is 0, then the output of the OR-gate appears at the device output, otherwise the device output is in floating state.

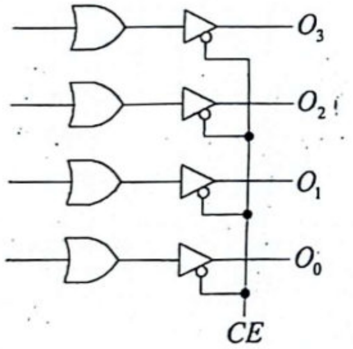


Figure 12.17 Output buffering of a PLA with 4 outputs.

The PLAs suitable for sequential logic applications have flip-flops and tri-state buffers in the outputs as shown in Figure 12.18. In this cases positive-edge triggered S-R flip-flops are connected at the outputs of the OR-gates, and the device outputs are available through tri-state buffers.

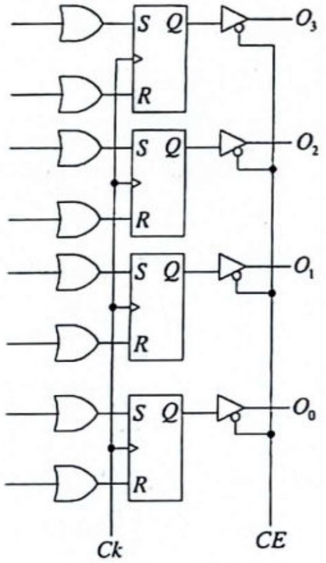


Figure 12.18 PLA OR-array with flip-flop in the output.

12.8.2 Boolean Function Implementation with Programmable Logic Array

Combinational logic implementation using PLA is illustrated using a combinational circuit, which is defined by the following four functions

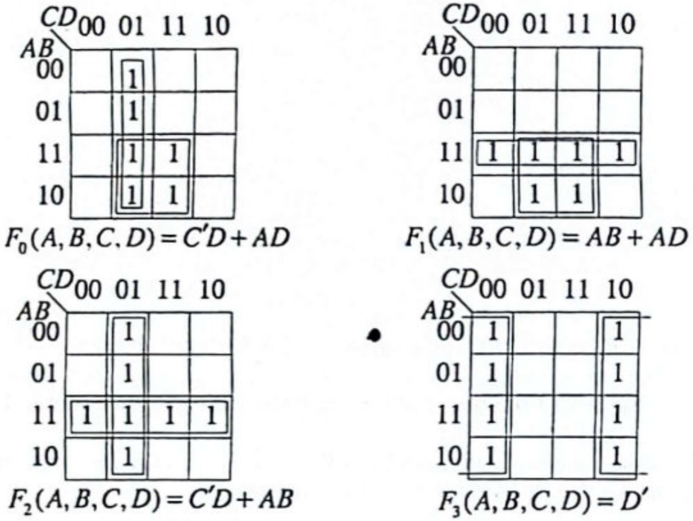
$$F_0(A,B,C,D)=\sum(1,5,9,11,13,15)$$
$$F_1(A,B,C,D)=\sum(9,11,12,13,14,15)$$
$$F_2(A,B,C,D)=\sum(1,5,9,12,13,14,15)$$
$$F_3(A,B,C,D)=\sum(0,2,4,6,8,10,12,14)$$

The combinational circuit is implemented with a PLA having 4 input, 6 product terms, and 4 outputs. If we want, as in the case of PROM, to implement the four functions as sum of minterms, 14 AND gates will be required. But the given PLA has only 6 AND gates. Therefore, the functions are to be simplified to accommodate in the given PLA. The four functions are simplified in the K-map of Figure 12.19. Each product term in the expression requires an AND gate. Since the number of AND gates in a PLA is finite, it is necessary to simplify the function to a minimum number of product terms in order to minimize the number of AND gates used. The functions are simplified in the K-map of Figure 12.19, which gives

$$F_0(A,B,C,D)=C'D+AD$$

$$F_1(A,B,C,D) = AB + AD$$
$$F_2(A,B,C,D) = C'D + AB$$
$$F_3(A,B,C,D) = D'$$

There are four distinct product terms in this combinational circuit: $C'D$, AD, AB, and D'.



$F_0(A,B,C,D) = C'D + AD$ $F_1(A,B,C,D) = AB + AD$

$F_2(A,B,C,D) = C'D + AB$ $F_3(A,B,C,D) = D'$

Figure 12.19 Simplification of functions using K-map.

Programming the PLA means that we specify the paths in its AND-OR pattern. The PLA program table is shown in Table 12.7. It consists of three columns. The first column lists the product terms numerically. The second column specifies the required paths between inputs and AND gates. The third column specifies the paths between the AND gates and the OR gates. Each product term is associated with an AND gate. The paths between the inputs and the AND gates are specified under the column heading *inputs*. A 1 in the input column specifies a path from the corresponding input to the input of the AND gate that forms the product term. A 0 in the input column specifies a path from the corresponding complemented input to the input of the AND gate. A dash specifies no connection. The appropriate fuses are blown and the ones left intact form the desired paths, as shown in Figure 12.20. It is assumed that the open terminals in the AND gate behaves like a 1 input.

Table 12.7 PLA program table.

	Product	Inputs $A\,B\,C\,D$	Outputs $F_3F_2F_1F_0$
$C'D$	1	- - 0 1	- 1 - 1
AD			- - 1 1
	2	1 - - 1	- 1 1 -
AB	3	1 1 - -	1 - - -
D'		- - - 0	- - - -
	4	- - - -	- - - -
	5	- - - -	
	6		

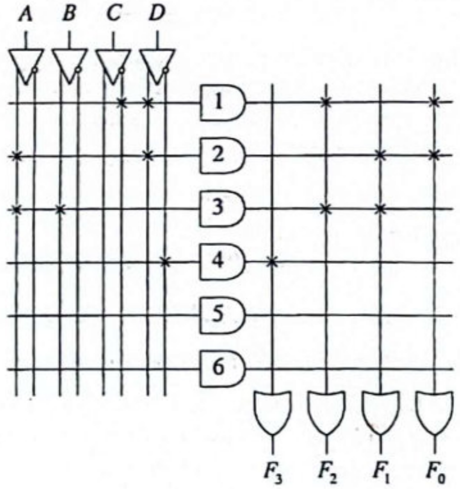


Figure 12.20 Implementation of functions using a PLA.

The path between the AND and OR gates are specified under the column heading *outputs*. The output variables are marked with 1s for all those product terms that formulate the function. We have

$$F_0(A,B,C,D) = C'D + AD$$

so F_0 is marked with 1s for product terms 1 and 2 and with dashes for product terms 3 through 6. Each product term that has a 1 in the output column requires a path from the corresponding AND gate to the output OR gate. Those marked with a dash specify no connection. The internal paths of the PLA for this circuit are shown in Figure 12.20. It is assumed that an open terminal in an OR gate behaves like a 0 input.

12.8.3 Available PLA Devices

Some of the available PLA devices are given in Table 12.8.

Table 12.8 Some available PLA devices.

IC No.	Inputs	Product terms	Outputs
PLS100	16	48	8
PLS105	16	48	8
82S200	16	48	8
82S201	16	48	8
82S100	16	48	8
82S101	16	48	8
DM7575	14	96	8
DM7576	14	96	8

12.8.4 PLA Based Logic Implementations

For combinational logic design PLAs with inverting/non-inverting outputs or tri-state buffered outputs are used. On the other hand, for sequential logic design PLAs having flip-flops in the outputs are used.

Combinational Logic Implementation

Implementation of the functions $F_0(A,B,C,D) = C'D + AD$, $F_1(A,B,C,D) = AB + AD$, $F_2(A,B,C,D) = C'D + AB$, and $F_3(A,B,C,D) = D'$ using the PLS100 PLA device is shown in Figure 12.21. The PLS100 PLA device has 16 inputs, 48 product terms, and 8 outputs. The outputs are inverting/non-inverting and tri-state buffered. The fuses of EXOR gates are to be kept intact for non-inverting outputs. The tri-state buffers are enabled through pin 19 of the IC and a 0 is required to be applied to this pin for enabling the output buffers.

Simplifications of the functions are shown in Figure 12.19 and the PLA program table for PLS100 PLA is shown in Table 12.9. The PLS100 PLA is programmed to implement the program table of Table 12.9 as shown in Figure 12.21. The inputs *A*, *B*, *C*, and *D* are applied at pins 9, 8, 7, and 6, respectively. The AND gates are numbered from 1 to 48 starting from the left most gate. Gates 1 through 4 are used for generation of the four product terms. The relevant connections are indicated in Figure 12.21 in the rows corresponding to the inputs. The output pins used for the outputs F_0 through F_3 are 18, 17, 16, and 15, respectively. The relevant connections are indicated in the figure. One input of each of the output EXOR gate is connected to ground through a fuse.

Table 12.9 PLA program table for the functions $F_0(A,B,C,D) = C'D + AD$, $F_1(A,B,C,D) = AB + AD$, $F_2(A,B,C,D) = C'D + AB$, and $F_3(A,B,C,D) = D'$ [Unused gates are not shown, which have all dashes in the *Inputs* and *Outputs* columns].

	Product	Inputs $A\,B\,C\,D$	Outputs $F_3F_2F_1F_0$
$C'D$	1	- - 0 1	- 1 - 1
AD	2	1 - - 1	- - 1 1
AB	3	1 1 - -	- 1 1 -
D'	4	- - - 0	1 - - -

Sequential Logic Implementation

As an example of sequential logic implementation using PLA, a ÷8 up/down counter is implemented using sequential PLA PLS105. The PLS105 PLA device has 16 inputs, 48 product terms, and 8 outputs. It also has 14 S-R flip-flops, 6 of which are buried within the device. The outputs of these 6 flip-flops are feedback to the AND-array. The remaining 8 S-R flip-flops are associated with the device outputs. Pin 1 is used for clock input and pin 19 for providing preset input (*P*) to the flip-flops and enable input to the output buffers.

The excitation table for ÷8 up/down counter using S-R flip-flop is shown in Table 12.10. The simplification of the flip-flop inputs using K-map is shown in Figure 12.22. The PLA program table for the flip-flop inputs is shown in Table 12.11 (unused AND gates are not shown).

The PLS105 PLA device is programmed for the program table of Table 12.11 as shown in Figure 12.23. The input *M* is applied at pin 9. AND gates 1 through 10 are used for generation of the ten product terms required for the six inputs of the three S-R flip-flops. The relevant connections are indicated in Figure 12.23 in the rows corresponding to the input *M* and three outputs of the used flip-flops. The output pins used for the counter outputs Q_2 through Q_0 are 18, 17, and 16, respectively. The relevant connections for the flip-flop inputs are indicated in the figure. Beside the output flip-flops, three internal flip-flops are programmed similar to the output flip-flops for feedback connections to the AND gate inputs.

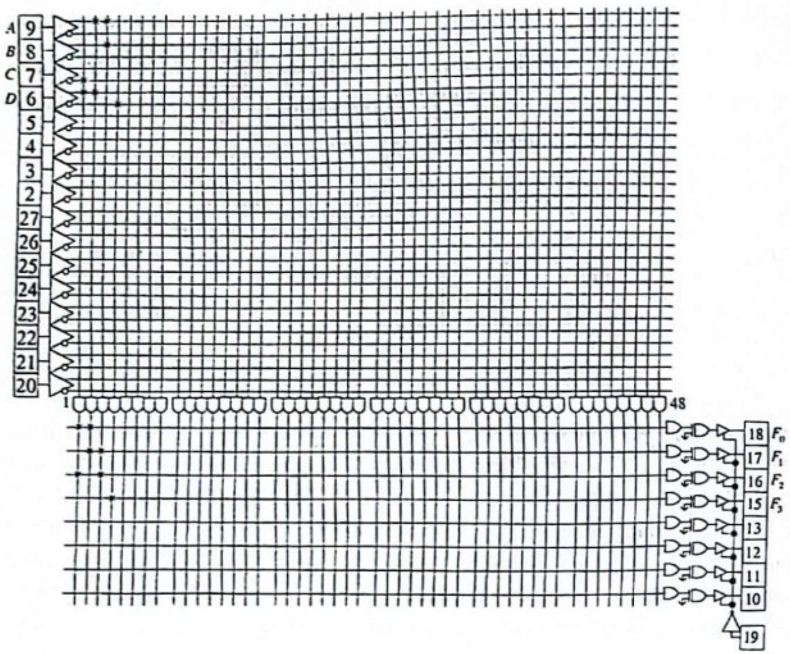
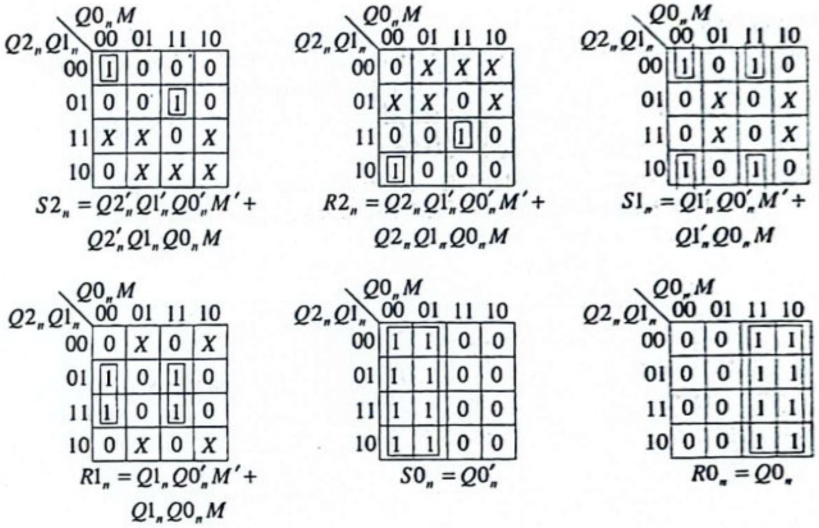


Figure 12.21 PLS100 PLA programmed for implementing functions $F_0(A,B,C,D) = C'D + AD$, $F_1(A,B,C,D) = AB + AD$, $F_2(A,B,C,D) = C'D + AB$, and $F_3(A,B,C,D) = D'$.

Table 12.10 Excitation table of a ÷8 up/down counter.

Present State	Next State		Flip-flop Inputs					
			$M=0$			$M=1$		
$Q2_nQ1_nQ0_n$	$Q2_{n+1}Q1_{n+1}Q0_{n+1}$ $M=0$	$M=1$	$S2_nR2_n$	$S1_nR1_n$	$S0_nR0_n$	$S2_nR2_n$	$S1_nR1_n$	$S0_nR0_n$
000	111	001	10	10	10	0X	0X	10
001	000	010	0X	0X	01	0X	10	01
010	001	011	0X	01	10	0X	X0	10
011	010	100	0X	X0	01	10	01	01
100	011	101	01	10	10	X0	0X	10
101	100	110	X0	0X	01	X0	10	01
110	101	111	X0	01	10	X0	X0	10
111	110	000	X0	X0	01	01	01	01



$Q2_nQ1_n \backslash Q0_nM$	00	01	11	10
00	1	0	0	0
01	0	0	1	0
11	X	X	0	X
10	0	X	X	X

$S2_n = Q2'_nQ1'_nQ0'_nM' + Q2'_nQ1_nQ0_nM$

$Q2_nQ1_n \backslash Q0_nM$	00	01	11	10
00	0	X	X	X
01	X	X	0	X
11	0	0	1	0
10	1	0	0	0

$R2_n = Q2_nQ1'_nQ0'_nM' + Q2_nQ1_nQ0_nM$

$Q2_nQ1_n \backslash Q0_nM$	00	01	11	10
00	1	0	1	0
01	0	X	0	X
11	0	X	0	X
10	1	0	1	0

$S1_n = Q1'_nQ0'_nM' + Q1'_nQ0_nM$

$Q2_nQ1_n \backslash Q0_nM$	00	01	11	10
00	0	X	0	X
01	1	0	1	0
11	1	0	1	0
10	0	X	0	X

$R1_n = Q1_nQ0'_nM' + Q1_nQ0_nM$

$Q2_nQ1_n \backslash Q0_nM$	00	01	11	10
00	1	1	0	0
01	1	1	0	0
11	1	1	0	0
10	1	1	0	0

$S0_n = Q0'_n$

$Q2_nQ1_n \backslash Q0_nM$	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	0	0	1	1
10	0	0	1	1

$R0_n = Q0_n$

Figure 12.22 Simplification of the flip-flop input functions of a ÷8 up/down counter.

Table 12.11 PLA program table for a ÷8 up/down counter.

	Product	Inputs $Q2_nQ1_nQ0_nM$	Outputs $S2_nR2_nS1_nR1_nS0_nR0_n$
$Q2'_nQ1'_nQ0'_nM'$	1	0 0 0 0	1 - - - - -
$Q2'_nQ1_nQ0_nM$	2	0 1 1 1	1 - - - - -
$Q2_nQ1'_nQ0'_nM'$	3	1 0 0 0	- 1 - - - -
$Q2_nQ1_nQ0_nM$	4	1 1 1 1	- 1 - - - -
$Q1'_nQ0'_nM'$	5	- 0 0 0	- - 1 - - -
$Q1'_nQ0_nM$	6	- 0 1 1	- - 1 - - -
$Q1_nQ0'_nM'$	7	- 1 0 0	- - - 1 - -
$Q1_nQ0_nM$	8	- 1 1 1	- - - 1 - -
$Q0'_n$	9	- - 0 -	- - - - 1 -
$Q0_n$	10	- - 1 -	- - - - - 1

Exercises

12.1 For a memory with M words, find the number of address lines and address range in binary format for each of the following cases:

(a) $M = 4$, (b) $M = 16$, (c) $M = 64$, (d) $M = 256$,

(e) $M = 1024$, (f) $M = 2048$

12.2 The block diagram of Intel 2141, a 1024 × 4 bit RAM, is shown in Figure 12.24. Implement the following RAMs using 2142 and other ICs:

(a) 4096 × 4 bit RAM

(b) 1024 × 8 bit RAM

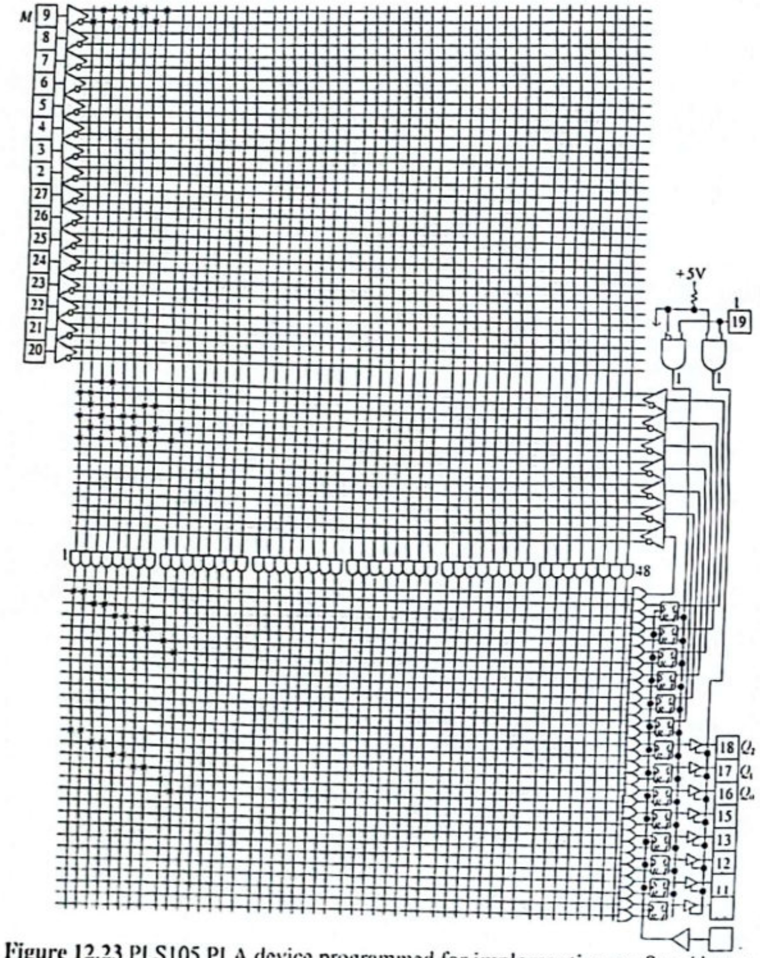

M
9
8
7
6
5
4
3
2
27
26
25
24
23
22
21
20
+5V
1
19
1
1
1
48
18
17
16
15
13
12
11

Figure 12.23 PLS105 PLA device programmed for implementing a ÷8 up/down counter.

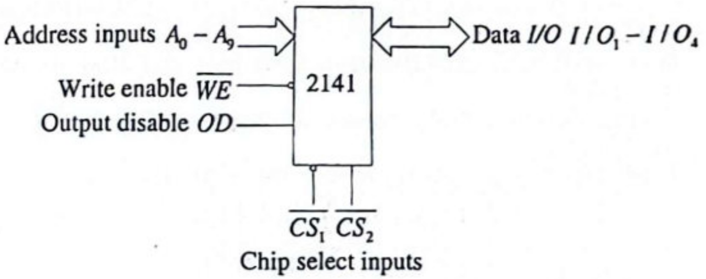

Address inputs $A_0 - A_9$
Data I/O $I/O_1 - I/O_4$
Write enable $\overline{WE}$
2141
Output disable OD
$\overline{CS_1}$ $\overline{CS_2}$
Chip select inputs

Figure 12.24 Block diagram of 2142 RAM.

12.3 Given a 32 × 8 ROM chip with an enable input, show the external connection necessary to construct a 128 × 8 ROM with four chips and a decoder.

12.4 The block diagram of Intel 2716 EPROM is shown in Figure 12.25. Implement the following ROMs using 2716 and other ICs

(a) 4096 × 8 ROM

(b) 2048 × 16 ROM

(c) 4096 × 16 ROM

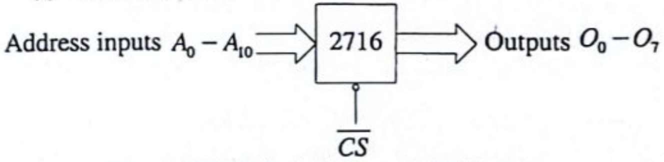

Address inputs $A_0 - A_{10}$
2716
Outputs $O_0 - O_7$
$\overline{CS}$

Figure 12.25 4Block diagram of 2816 EPROM.

12.5 Specify the size of a ROM (number of words and number of bits per word) that will accommodate the truth table for the following combinational circuit components:

(a) A binary multiplier that multiplies two 4-bit numbers.

(b) A 4-bit adder/subtractor.

(c) A quadruple 2 × 1 multiplexers with common select and enable inputs.

12.6 Tabulate the truth table for an 8 × 4 ROM that implements the following four Boolean functions:

$$F_0(x, y, z) = \sum(1,2,4,6)$$

$$F_1(x, y, z) = \sum(0,1,6,7)$$

$$F_2(x, y, z) = \sum(2,6)$$

$$F_3(x, y, z) = \sum(1,2,3,5,7)$$

12.7 Design a BCD-to-Excess-3 code converter using (a) PROM, (b) PAL, and (c) PLA.

12.8 Design an Excess-3-to-BCD code converter using (a) PROM, (b) PAL, and (c) PLA.

12.9 Design a BCD-to-seven segment decoder using (a) PROM, (b) PAL, and (c) PLA.

12.10 Tabulate the PLA programming table for the four Boolean functions:

$$F_0(x, y, z) = \sum(1,2,4,6)$$

$$F_1(x, y, z) = \sum(0,1,6,7)$$

$$F_2(x, y, z) = \sum(2,6)$$

$$F_3(x, y, z) = \sum(1,2,3,5,7)$$

12.11 Derive the PLA programming table for the combinational circuit that squares a 3-bit number. Minimize the number of product terms.

Further Reading

1. Richard S. Sandige, *Modern Digital Design*, McGraw-Hill Publishing Company, 1990.
2. R. P. Jain, *Modern Digital Electronics*, 2nd ed., Tata McGraw-Hill Publishing Company Limited, India, 1998.
3. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
4. D. C. Green, *Digital Electronics*, 5th ed., Pearson Education, 2001.
5. Ronald J. Tocci and Neal S. Widmer, *Digital Systems: Principles and Applications*, 8th ed., Prentice-Hall, Inc., 2001.
6. M. Morris Mano, *Digital Design*, 3rd ed., Pearson Education, 2002.
7. Samuel C. Lee, *Digital Circuits and Logic Design*, Prentice-Hall, Inc., 2002.

CHAPTER 13
REGISTER TRANSFER LOGIC

13.1 Introduction

A digital system is a sequential logic circuit constructed with flip-flops and gates. Normally, sequential circuits are specified with state tables as shown in Chapter 10. Specifying a large digital system with a state table is very difficult, because the number of states would be very large. To overcome this difficulty, digital systems are designed using a modular approach. The system is partitioned into modular subsystems, each of which performs some functional task. The modules are constructed from digital devices like registers, decoders, multiplexers, arithmetic elements, and control logic. The various modules are interconnected with common data and control paths to form a digital system. These modules are described by a set of registers and the operations performed on the information stored in them. The information flow and processing performed on the data stored in the registers are referred to as register transfer operations. A digital system designed around registers is referred to as the *register transfer logic* (RTL).

In this chapter we will discuss digital system design at the RTL level.

13.2 Basics of Register Transfer Logic

A register transfer logic (RTL) level digital system is specified by the following three components:

1. A set of registers in the system.
2. The operations performed on the data stored in the registers.
3. The control for supervising the sequence of operations in the system.

A register is a group of flip-flops that stores binary information and has the capability of performing one or more elementary operations like shift, count, clear, and load. A register can load new information or shift the information to the right or to the left. A counter is considered as register which increments or decrements a number by one. A flip-flop standing alone is considered as a 1-bit register that can be set, cleared, or complemented. In fact, the flip-flops and associated gates of any sequential circuit can be considered as register by this definition.

The operations executed on the information stored in registers are elementary operations that are performed in parallel on a string of bits during one clock cycle. The result of the operation may replace the previous information in a register. Alternatively, the result may be transferred to another register, leaving the previous data unchanged. A presetable counter is able to perform the increment by one and load operations. A bidirectional shift register is able to perform the shift right and shift left operations.

The control that initiates the sequence of operations consists of timing signals that sequence the operations in a prescribed manner. Certain conditions that depend on results of previous operations may determine the sequence of future operations. The outputs of the control logic are binary variables that initiate the various operations in the register.

Information transfer from one register to another is described in symbolic form by means of a replacement operator. The statement

$$R2 \leftarrow R1$$

denotes a transfer of the contents of register $R1$ into register $R2$. It specifies a replacement of the contents of $R2$ by the contents of $R1$. By definition, the contents of the source register $R1$ do not change after the transfer. Such register transfer statement implies that circuit connections are available from the outputs of the source register to the inputs of the destination register and that the destination register has a parallel load capability.

Normally, the register transfer occurs under a predetermined condition. A conditional statement is described with an if-then statement such as

$$\text{If } (T1 = 1) \text{ then } (R2 \leftarrow R1)$$

Where $T1$ is a control signal generated in the control section. It is assumed that all transfers occur during a clock-edge transition. Although a control condition such as $T1$ may become true before the clock transition, the actual transfer does not occur until the clock transition is completed. Two or more operations that are executed at the same time are separated by a comma. The statement

$$\text{If } (T3 = 1) \text{ then } (R2 \leftarrow R1,\ R1 \leftarrow R2)$$

denotes an operation that exchanges the contents of two registers during the same clock edge, provided that $T3 = 1$. This simultaneous operation is possible with registers that have edge-triggered flip-flops. Some of the register transfers are as follows:

$R1 \leftarrow R1 + R2$	Add content of $R2$ to $R1$
$R3 \leftarrow R3 + 1$	Increment $R3$ by 1
$R4 \leftarrow \text{shr } R4$	Shift right $R4$
$R5 \leftarrow 0$	Clear $R5$ to 0

The addition is done with a binary parallel adder. The incrementing is done with a counter. The shift is done with a shift register.

The type of operations most often encountered in digital systems can be classified into four categories:

1. Transfer operations that transfer data from one register to another.
2. Arithmetic operations that perform arithmetic on data in registers.
3. Logic operations that perform bit manipulation of non-numeric data in registers.
4. Shift operations that shift data in registers.

The transfer operation does not change the information content of the data being moved from the source register to the destination register. The other three types change the information content during the transfer.

The information stored in a digital system can be classified as data and control information. Data are information that are manipulated to perform arithmetic, logic, shift, and other data processing tasks. These data processing operations are implemented with adders, decoders, multiplexers, counters, and shift registers. Control information provides command signals for supervising the various operations in the data section in order to accomplish the desired data processing tasks. The logic design of a digital system can be divided into two distinct parts. One part is the design of the digital circuits that perform the data processing operations. The other part is the design of the control circuits that determines the sequence in which the various actions are performed.

Figure 13.1 shows the relationship between the control logic and the data processing in a digital system. The data processing path is commonly referred to as the *datapath*, which manipulates data in registers according to the system's requirements. The control logic initiates a sequence of commands to the datapath. The control logic uses status conditions from the datapath to serve as decision variables for determining the sequence of control signals.

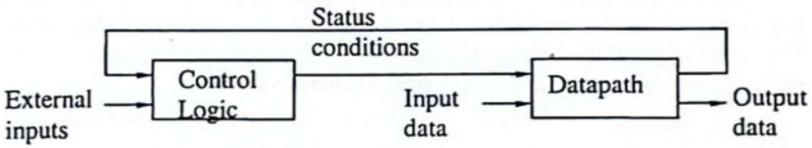


Figure 13.1 Relationship between control and datapath.

The control logic that generates the signals for sequencing the operations in the datapath is a sequential circuit whose internal states dictate the control commands for the system. At any given time, the state of the sequential control initiates a prescribed set of commands. Depending on status conditions and other external inputs, the sequential control goes to the next state to initiate other operations. The digital circuits that act as the control logic provide a time sequence of signals for initiating the operations in the datapath and also determine the next state of the control subsystem itself.

13.3 Algorithmic State Machines

13.3.1 The ASM Chart

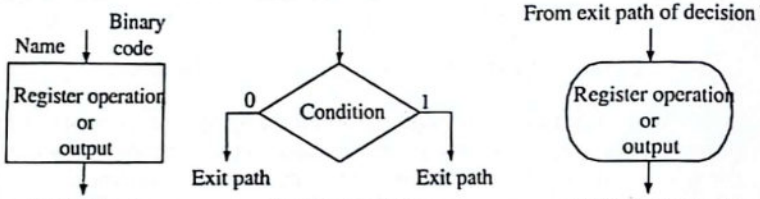
The control sequence and datapath tasks of a digital system are specified by means of a hardware algorithm. A hardware algorithm is a procedure for implementing the problem with a given piece of equipment. The most challenging and creative part of digital design is the formulation of hardware algorithms for achieving required objectives. A special flowchart that has been developed to define digital hardware algorithms is called an ***algorithmic state machine*** (ASM) chart. A **state machine** is another term for a sequential circuit, which is the basic structure of a digital system. The ASM chart describes the sequence of events as well as the timing relationship between the states of a sequential controller and the events that occur while going from one state to the next. It is specially developed to specify accurately the control sequence and datapath operations in a digital system, taking into consideration of the constraints of digital hardware.

The ASM chart is a special type of flowchart suitable for describing the sequential operations of a digital system. The chart is composed of three basic elements: the state box, the decision box, and the condition box.

A *state box* indicates a state in the control sequence as shown in Figure 13.2(a). The shape of the state box is a rectangle within which are written register operations or output signal names that the control generates while being in this state. The state is given a symbolic name, which is placed at the upper left corner of the box. The binary code assigned to the state is placed at the upper right corner.

The *decision box* describes the effect of an input on the control subsystem. It has a diamond-shaped box with two or more exit paths, as shown in Figure 13.2(b). The input condition to be tested is written inside the box. One exit path is taken if the condition is true and another when the condition is false. When an input condition is assigned a binary value, the two paths are indicated by 1 and 0.

The *conditional box* has an oval shape as shown in Figure 13.2(c). The input path to the conditional box must come from one of the exit paths of a decision box. The register operations or outputs listed inside the conditional box are generated during a given state, provided that the input condition is satisfied.



(a) State box (b) Decision box (c) Conditional box

Figure 13.2 Elements of the ASM chart.

A segment of an ASM chart is shown in Figure 13.3. The meaning of the ASM chart is that the control generates a START output signal when in state *T1*. While in state *T1*, the control checks the status of input *E*. If *E* = 1, then *R* is cleared to 0; otherwise, *R* remains unchanged. In either case, the next state is *T2*.

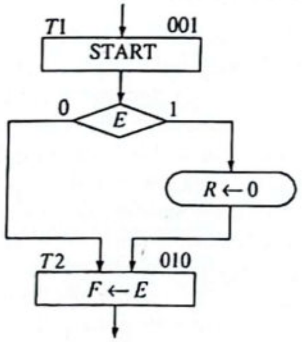


Figure 13.3 A segment of an ASM chart.

An *ASM block* is a structure consisting of one state box and all the decision and conditional boxes connected to its exit paths. An ASM block has one entrance and any number of exit paths represented by the structure of the decision boxes. An ASM chart consists of one or more interconnected blocks. An example of an ASM block is shown in Figure 13.4. Associated with state *T1* are two decision boxes and one conditional box. The portion enclosed within the dotted box is an ASM block. A state box without any decision or conditional boxes constitutes a simple block Each block in the ASM chart describes the state of the system during one clock-pulse interval. The operations within the state and conditional boxes in Figure 13.4 are executed with a common clock pulse while the system is in state *T1*. The same clock pulse also transfers the system controller to one of the next states – *T2*, *T3*, or *T4* – as dictated by the binary values of *E* and *F*.

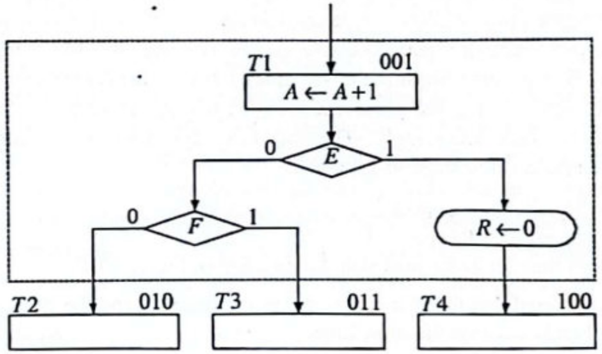


Figure 13.4 An ASM block.

The ASM chart is very similar to a state diagram. Each state block is equivalent to a state in a sequential circuit. The decision box is equivalent to the binary information written along the directed edges that connect two states in a state diagram. As a consequence, it is sometimes convenient to convert the ASM chart into a state diagram and then use sequential circuit procedures to design the control logic. As an illustration, the ASM chart of Figure 13.4 is drawn as a state diagram in Figure 13.5. The three states are indicated by circles, with their binary values written inside the circles. The directed lines indicate the conditions that determine the next state. The unconditional and conditional operations that must be performed are not indicated in the state diagram.

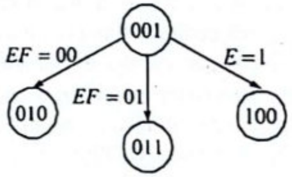


Figure 13.5 State diagram equivalent to the ASM chart of Figure 13.4.

13.3.2 Timing Considerations

The timing for all registers and flip-flops in a digital system is controlled by a master-clock generator. The clock pulses are applied not only to the registers of the datapath, but also to all the flip-flops in the control logic. Inputs are also synchronized with the clock because they are normally generated as outputs of another circuit that uses the same clock signal. If the input signal changes at an arbitrary time independent of the clock, we call it an asynchronous input. Asynchronous inputs may cause a variety of problems. To simplify the design, we assume that all inputs are synchronized with the clock and change state in response to an edge transition.

An ASM chart considers the entire block as one unit. All the operations that are specified within the block must occur in synchronism during the edge transition of the same clock pulse while the system changes from *T1* to the next state. This is presented pictorially in Figure 13.6. We assume positive-edge triggering of all flip-flops. The first positive transition of the clock transfers the control circuit into state *T1*. While in state *T1*, the control circuits check inputs *E* and *F* and generate appropriate signals accordingly. The following operations occur simultaneously during the next positive edge of the clock:

1. Register A is incremented.
2. If $E = 1$, register *R* is cleared.
3. Control transfer to the next state as specified in Figure 13.4.

It should be noted that the two operations in the datapath and the change of state in the control logic occur at the same time.

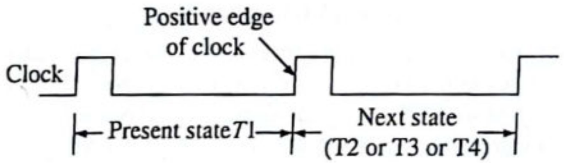


Figure 13.6 Transition between states.

13.4 Binary Multiplier

We will now demonstrate the RTL design procedure by designing a binary multiplier. In this section we will present a hardware algorithm for binary multiplication, propose the register configuration for its implementation, and then show the design of the datapath and control by using an ASM chart. We will design a binary multiplier which multiplies two unsigned binary numbers.

The multiplication of two binary numbers was discussed in Chapter 2. It is done with paper and pencil by successive additions and shifting. This process is illustrated with the following numerical example:

```
 23          10111        multiplicand
 19          10011        multiplier
---         ------
             10111
            10111
           00000
          00000
         10111
        ----------
437      110110101        product
```

The process consists of looking at successive bits of the multiplier starting from the least significant bit. If the multiplier bit is 1, the multiplicand is copied down; otherwise, 0s are copied down. The numbers copied in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product. The product obtained from the multiplication of two *n*-bit binary numbers can be up to 2*n* bits.

When the multiplication procedure is implemented with digital hardware, it is convenient to change the process slightly. The product is first assumed to be 0. Then the successive bits of the multiplier are checked starting from the least significant bit. If the multiplier bit is 1, then the multiplicand is added with the product with the most significant bits aligned together. Then the partial product is shifted 1-bit right. This is equivalent to shifting the multiplicand 1-bit left and adding it with the partial product. If the multiplier bit is 0, then the partial product is only shifted 1-bit right.

13.4.1 Datapath

The block diagram of the binary multiplier is shown in Figure 13.7. The multiplicand is stored in register *B*, the multiplier is stored in register *Q*, and the partial product is formed in register *A* and stored in *A* and *Q*. Register *A* is initially set to 0. A parallel adder adds the contents of register *A* and *B* and the result is stored in *A*. The *C* flip-flop stores the carry after the addition. The *P* counter is initially set to hold a binary number equal to the number of bits in the multiplier. This counter is decremented after the formation of each partial product. When the content of the counter reaches 0, the product is formed in the double register *A* and *Q* and the process stops. The control logic stays in an initial state until the start signal *S* becomes 1. The system then performs the multiplication. The sum of *A* and *B* forms the *n* most significant bits of the partial product, which is transferred to *A*. The output carry from the addition, whether 0 or 1, is transferred to *C*. Both the partial product in *A* and the multiplier in *Q* are shifted to the right. The least significant bit of *A* is shifted into the most significant position of *Q*; the carry from *C* is shifted into the most significant position of *A*; and a 0 is shifted into *C*. After the shift right operation, one bit of the partial product is transferred into *Q* while the multiplier bits in *Q* are shifted one position to the right. In this manner, the least significant bit of register Q, designated by Q_0, holds the bit of the multiplier that must be checked next. The control logic determines whether to add or not, based on this input bit. The control logic also receives signal Z from a circuit that checks counter P for 0. Q_0 and Z are status inputs for the control unit. The start input S is an external control input. The outputs of the control logic activate the required operations in the registers.

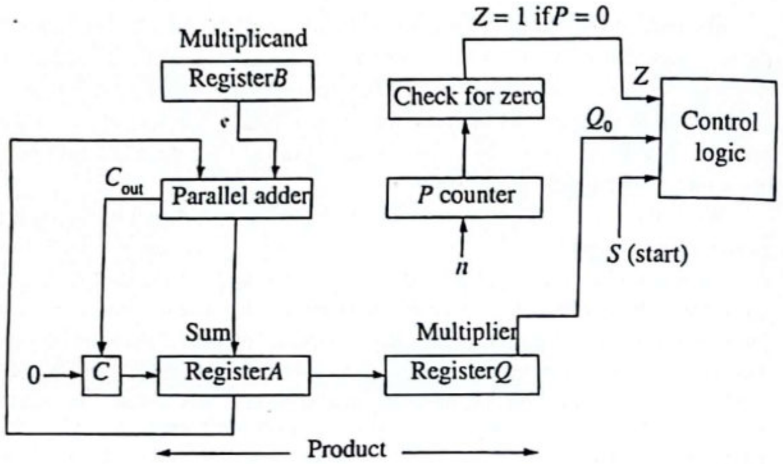

Multiplicand
Register B
Z = 1 if P = 0
Check for zero
Z
Q_0
Control logic
C_{out}
Parallel adder
P counter
n
S (start)
Sum
Multiplier
0
C
Register A
Register Q
Product

Figure 13.7 Block diagram of binary multiplier.

13.4.2 ASM Chart

The ASM chart for the binary multiplier is shown in Figure 13.8. Initially, the multiplicand is in register B and the multiplier is in register Q. As long as the circuit is in the initial state and $S = 0$, no action occurs and the system remains in an initial state $T0$. The multiplication process starts when $S = 1$ and control goes to state $T1$. Register A and carry flip-flop C are cleared to 0 and the sequence counter P is set to a binary number n, equal to the number of bits in the multiplier. The system then goes to state $T2$. The multiplier bit in Q_0 is checked, and if it is equal to 1, the multiplicand in B is added to the partial product in A. The carry from the addition is transferred to C. The partial product in A and C is left unchanged if $Q_0 = 0$. The P counter is decremented by 1 regardless of the value of Q_0. In both cases, the next state is $T3$. Registers C, A, and Q are combined into one composite register CAQ, and its contents are shifted once to the right to obtain a new partial product.

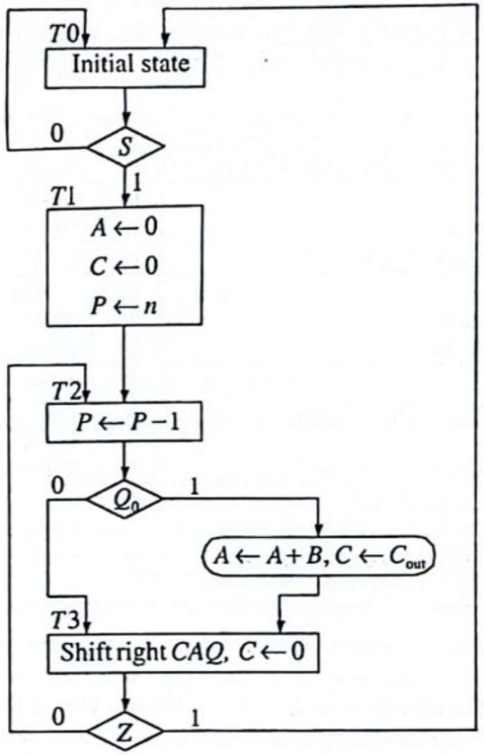

T0
Initial state
0
S
1
T1
$A \leftarrow 0$
$C \leftarrow 0$
$P \leftarrow n$
T2
$P \leftarrow P-1$
0
Q_0
1
$A \leftarrow A+B, C \leftarrow C_{out}$
T3
Shift right CAQ, $C \leftarrow 0$
0
Z
1

Figure 13.8 ASM chart for binary multiplier.

The value in the P counter is checked after the formation of each partial product. If the content of P is not 0, status bit Z is equal to 0 and the process is repeated to form a new partial product. The process stops when the P counter reaches 0 and the control input Z is equal to 1. The partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in A and Q, with A holding the most significant bits and Q the least significant bits of the product.

The previous numerical example is repeated in Table 13.1 to clarify the multiplication process. The procedure follows the steps outlined in the ASM chart.

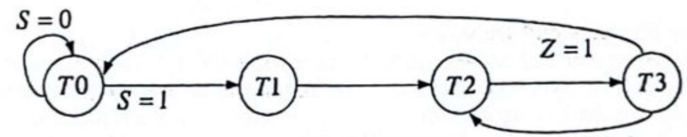
Table 13.1 Numerical example for binary multiplier.

Multiplicand $B = 10111$	C	A	Q	P
Multiplier in Q	X	XXXXX	10011	XXX
Multiplication starts, $C = 0, A = 0, P = 101$	0	00000	10011	101
Decrement P	0	00000	10011	100
$Q_0 = 1$, add B to A	0	10111	10011	100
Shift right CAQ, set C to 0	0	01011	11001	100
Decrement P	0	01011	11001	011
$Q_0 = 1$, add B to A	1	00010	11001	011
Shift right CAQ, set C to 0	0	10001	01100	011
Decrement P	0	10001	01100	010
$Q_0 = 0$, shift right CAQ, set C to 0	0	01000	10110	010
Decrement P	0	01000	10110	001
$Q_0 = 0$, shift right CAQ, set C to 0	0	00100	01011	001
Decrement P	0	00100	01011	000
$Q_0 = 1$, , add B to A	0	11011	01011	000
Shift right CAQ, set C to 0	0	01101	10101	000
Final product $AQ = 0110110101$				

The type of registers needed for the data processing subsystem can be derived from the register operations listed in the ASM chart. Register A is a parallel-in parallel-out right-shift register and has a synchronous clear capability to reset the register to 0. It loads the result from the adder in parallel and outputs the contents in parallel to add with B in the adder. Initially its value is set to 0 by reset. Register Q is a parallel-in serial-out right-shift register. It loads the multiplier in parallel. The counter P is binary down counter with a facility to parallel load a binary constant equal to the number of bits in the multiplier. The C flip-flop must be designed to accept the input carry and have a synchronous clear. Register B is a parallel-in parallel-out register. It loads the multiplicand in parallel and outputs the contents in parallel to the adder.

13.4.3 Control Logic

The control logic design is a sequential circuit design problem. It may be convenient to formulate the state diagram of the sequential control from the ASM chart. The control state diagram for the binary multiplier developed from the ASM chart of Figure 13.8 is shown in Figure 13.9(a). The four states $T0$ through $T3$ correspond to the state boxes of Figure 13.8. The inputs S and Z correspond to the decision boxes of Figure 13.8. The register transfer operations for each of the four states are listed in Figure 13.9(b). They correspond to the state and conditional boxes of Figure 13.8.



(a) State diagram

$T0:$ Initial state

$T1: A \leftarrow 0, C \leftarrow 0, P \leftarrow n$

$T2: P \leftarrow P - 1$

$\quad$ if $(Q_0 = 1)$ then $(A \leftarrow A + B, C \leftarrow C_{out})$

$T3:$ shift right $CAQ, C \leftarrow 0$

(b) Register transfer operations

Figure 13.9 Control specification for binary multiplier.

There are two distinct aspects of implementing the control logic: (i) establishing the required sequence of states and (ii) providing signals to control the register operations. The sequence of states is specified in the state diagram. The signals for controlling the operations in the registers are specified in the register transfer statements. For the binary multiplier, the signals are $T1$ (for clearing A and C and loading a number into P), $T2$ (for decrementing P), $T3$ (for shifting register CAQ), and Q_0 to determine whether B is added to A or not. The block diagram of the control is shown in Figure 13.10. The inputs for the sequential control are S and Z, and the outputs are $T0$, $T1$, $T2$, and $T3$, as specified in the state diagram of Figure 13.9(a). The AND gate produces $L = T2Q_0$, which loads the sum into register A if $Q_0 = 1$ while in state $T2$.

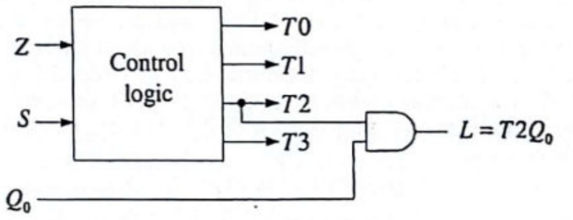


Figure 13.10 Control block diagram for binary multiplier.

As the control is a sequential circuit, it can be designed by the sequential logic design method discussed in Chapter 10. However, in most cases this method is difficult to carry out because a typical control circuit has a large number of states and inputs. To overcome this problem, specialized methods for control logic design are used. In this section we will introduce three such methods.

Sequence Register and Decoder

The sequence register and decoder method uses a sequence register for the control states and a decoder to provide an output corresponding to each of the states. An *n*-bit sequence register is a circuit with *n* flip-flops together with the associated gates that effect their state transition. The control state diagram of Figure 13.9(a) has four states and two inputs. To implement it with a sequence register and decoder, we need two flip-flops and a 2×4 decoder.

We designate the two required flip-flops as *G*1 and *G*0 and encode the four states *T*0, *T*1, *T*2, and *T*3 by ordinary binary numbers 00, 01, 10, and 11, respectively. Assuming this flip-flop designation and state encoding, the state table for the sequential control is shown in Table 13.2. It is derived directly from the state diagram of Figure 13.9(a). The input columns have don't care entries whenever the input variable is not used to determine the next state. The outputs of the control circuit are designated by the state names. The particular output variable that is equal to 1 at any given time is determined from the equivalent binary value of the present state. Thus, when the present state is *G*1*G*0 = 00, output *T*0 must be equal to 1 and the other outputs remain at 0. As the outputs are a function of only the present state, they can be generated with a decoder circuit having the two inputs *G*1 and *G*0 and the four outputs *T*0 through *T*3.

Table 13.2 State table for control circuit of binary multiplier.

Present state *G*1*G*0	Input *SZ*	Next state *G*1*G*0	Outputs *T*0*T*1*T*2*T*3
00	0X	00	1000
00	1X	01	1000
01	XX	10	0100
10	XX	11	0010
11	X0	10	0001
11	X1	00	0001

Instead of using flip-flop outputs as the present state conditions, we use the decoder outputs as present state conditions for the sequential circuit. We can now obtain the flip-flop input equations directly from inspection of the state table. For example, from the next state conditions in the state table of Table 13.3, we find that the next state of *G*1 is equal to 1 when the present state is *T*1, or when the present state is *T*2, or when the present state is *T*3 with *Z* = 0. These conditions can be specified by the equation:

$$D1 = T1 + T2 + T3Z'$$

where *D*1 is the *D* input of the flip-flop *G*1. Similarly, the *D* input of the flip-flop *G*0 can be determined as

$$D0 = T0S + T2$$

The logic diagram of the control circuit is shown in Figure 13.11. It consists of a sequence register with two flip-flops *G*1 and *G*0 and a 2×4 decoder with acive-1 outputs. The outputs of the decoder are used to generate the inputs to the next state logic as well as the control outputs. The outputs of the controller should be connected to the datapath to activate the required register operations.

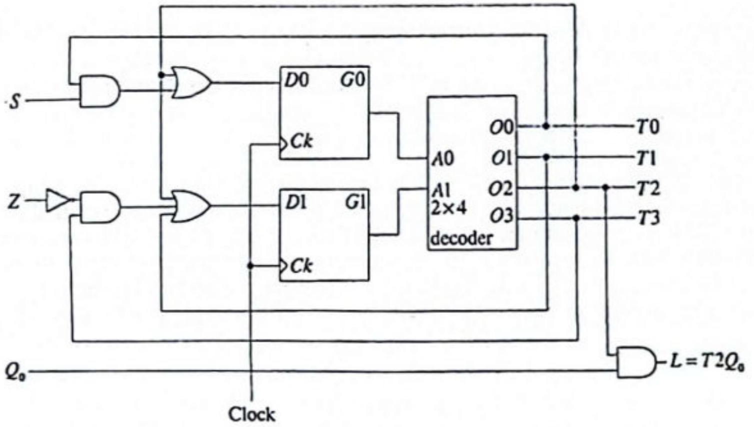


Figure 13.11 Logic diagram of control for binary multiplier using a sequence register and decoder.

One Flip-Flop per State

Another possible method of control logic design is the use of the one-hot assignment. This assignment uses as many bits as there are states in the circuit. At any given time, only one bit is equal to 1 (the one hot) and the all others are kept at 0 (all cold). Table 13.3 shows one-hot assignment for four states. This type of assignment uses a flip-flop for each state. Only one of the flip-flops contains a 1 at any time; all others are reset to 0. The single 1 propagates from one flip-flop to another under the control of decision logic. In such a configuration, each flip-flop represents a state that is present only when the control bit is transferred to it.

Table 13.3 One-hot assignment of four states.

State	One-hot assignment
*T*0	0001
*T*1	0010
*T*2	0100
*T*3	1000

This method uses the maximum number of flip-flops for the sequential circuit, one for each state. At first glance, it may seem that this method would increase system cost since more flip-flops are used. But the method offers some advantages over the sequence register and decoder method. One advantage is the simplicity with which the logic can be designed from inspection of the ASM chart or the state diagram. No state or excitation tables are needed if *D* flip-flops are used. It offers a savings in design effort, an increase in operational simplicity, and a possible decrease in the total number of gates since a decoder is not needed.

The design procedure is illustrated by obtaining the control circuit specified by the state diagram of Figure 13.9(a). Since there are four states in the state diagram, we choose four *D* flip-flops and label their outputs *T*0, *T*1, *T*2, and *T*3. The input equations for setting each flip-flop to 1 is determined from the present state and the input conditions along the corresponding directed lines going into the state. For example, flip-flop $T0$ is set to 1 with the next clock edge if the circuit is in present state $T3$ and input Z is equal to 1 or if the circuit is in present state $T0$ and S is equal to 0. These conditions are specified by the input equation:

$$D0 = T0S' + T3Z$$

where $D0$ is the input of flip-flop $T0$. In general, the condition for setting a flip-flop to 1 is obtained directly from the state diagram from the condition specified in the directed lines going into the corresponding flip-flop state ANDed with the previous flip-flop state. If there are more than one directed lines going into a state, all conditions must be ORed. Using this procedure for the other three flip-flops, we obtain the input equations:

$$D1 = T0S$$
$$D2 = T1 + T3Z'$$
$$D3 = T2$$

The logic diagram of the control is shown in Figure 13.12. It consists of four D flip-flops $T0$ through $T3$ and the associated gates specified by the input equations. Initially, flip-flop $T0$ must be set to 1 and all other flip-flops must be reset to 0 so that the flip-flops representing the initial state is enabled. This can be done by using an asynchronous preset on flip-flop $T0$ and an asynchronous clear for the other flip-flops. Once started, the one flip-flop per state will propagate from one state to the other in the proper manner. Only one flip-flop will be set to 1 with each clock edge; all others are reset to 0 because their D inputs are equal to 0.

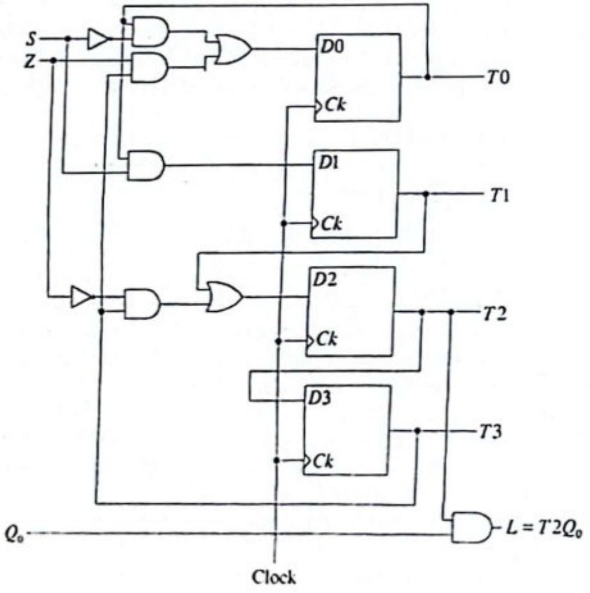


Figure 13.12 Logic diagram of control for binary multiplier using one flip-flop per state.

Multiplexer, Sequence Register, and Decoder

The sequence register and decoder control consists of three components: (i) the flip-flops that hold the binary state value, (ii) the decoder that generates the control outputs, and (iii) the gates that determine the next state. In Chapter 8, it was shown that a combinational circuit can be implemented using multiplexer. Replacing the gates with multiplexer results in a regular pattern of three levels of components. The first level consists of multiplexers that determine the next state of the register. The second level contains a register that holds the present state. The third level has a decoder that provides a separate output for each control state. These three components are predefined standard cells in many integrated circuits.

Let us consider the ASM chart of Figure 13.8. It consists of four states and two control inputs. We assume that the states $T0$, $T1$, $T2$, $T3$ are assigned the binary values 00, 01, 10, and 11, respectively. The decision boxes (except with Q_0) specify the state transitions as a function of the two control inputs - S and Z.

The three-level control implementation of the ASM chart of Figure 13.8 is shown in Figure 13.13. As the states have two bits, we need two multiplexers - MUX1 and MUX0, and a register with two flip-flops - $G1$ and $G0$. We also need a decoder with four outputs. The outputs of the register are applied to the decoder inputs and also to the address inputs of the multiplexers. In this way, the present state of the register is used to select one of the inputs from each multiplexer. The outputs of the multiplexers are then applied to the D inputs of $G1$ and $G0$. The purpose of each multiplexer is to produce an input to its corresponding flip-flip equal to the binary value of the next state. The inputs of the multiplexers are determined from the decision boxes and state transitions given in the ASM chart.

To facilitate the evaluation of the multiplexer inputs, we prepare a table showing the input conditions for each possible transition in the ASM chart. Table 13.4 shows this information for the ASM chart of Figure 13.8. The table has four columns - the first column lists the present state, the second column lists the corresponding next state, the third column lists the input condition for this state transition, and the fourth column lists the multiplexer inputs required for generating the next state. There are two transitions from present state 00 - it goes to next state 00 when the input S is 0 and goes to next state 01 when the input S is 1. This input information is listed under the input condition column. From the next state column we see that the next value of $G1$ is 0 for both the transitions. So, the input I_0 of MUX1 will be 0. For both the state transitions, the next state value of $G0$ is exactly equal to the input S. So, input I_0 of MUX0 will be S. These multiplexer inputs are listed under Inputs column. In the multiplexer input condition table each present state is separated by a horizontal line. From the ASM chart we see that the present state 01 always goes to the next state 10 independent of the control inputs. As the next state value of $G1$ is 1, input I_1 of MUX1 will be 1. Similarly, as the next state vale of $G0$ is 0, input I_1 of MUX0 will be 0. The present state 10 always goes to the next state 11 and the multiplexer input I_2 of both MUX1 and MUX0 will be 1. The present state 11 goes to the next state 10 when the input Z is 0 and goes to the next state 00 when the input Z is 1. The next state of $G1$ is complement of Z, so I_3 input of MUX1 will be Z'. The next state of $G0$ is always 0, so input I_3 of MUX0 will be 0. These information are listed in Table 13.4.

Table 13.4 Multiplexer input conditions for binary multiplier.

Present State	Next state	Input	Inputs	
$G1G0$	$G1G0$	Condition	MUX1	MUX0
00	00	S'	0	S
00	01	S'		
01	10		1	0
10	11		1	1
11	10	Z'	Z'	0
11	00	Z		

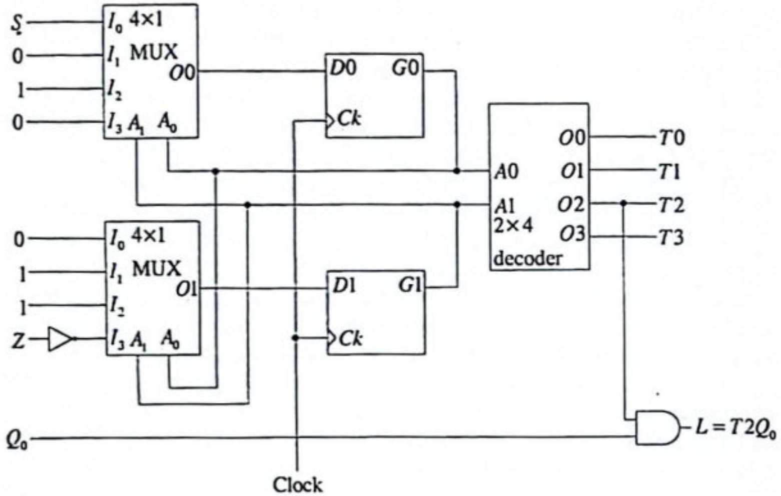


Figure 13.13 Logic diagram of control for binary multiplier using multiplexer, sequence register, and decoder.

13.5 Binary Divider

In this section we will illustrate the RTL-level design of a binary divider, which divides an unsigned binary number by another unsigned binary number. Binary division was discussed in Chapter 2. It is done with paper and pencil by successive subtractions and shifting. This process is illustrated with the following numerical example.

```
            101  Quotient
Divisor 101 | 11011 Dividend
            |-101
            -----
              111
             -101
            -----
               10  Remainder
```

In this process, we first place the divisor below the dividend left aligned. Then we see whether the divisor can be subtracted from the corresponding bits of the dividend. If subtraction is possible, then we do it to generate the remainder and place a 1 at the corresponding bit position of the quotient. Otherwise, the corresponding dividend bits form the remainder and we place a 0 at the corresponding bit position of the quotient. Then we append the next low-order bit of the dividend with the remainder, shift the divisor 1-bit right, and repeat the process. If an n-bit binary number is divided by another n-bit binary number, then the quotient will be up to n-bit.

When the division procedure is implemented with digital hardware, it is convenient to change the process slightly. The remainder is first assumed to be 0. Then the dividend is placed at the right of the remainder to form a $2n$-bit number. The $2n$-bit number is then shifted 1-bit left so that the most significant bit of the dividend becomes the least significant bit of the remainder. The divisor is then subtracted from the remainder. If subtraction is possible, then we do it to update the remainder and place a 1 at the corresponding bit position of the quotient. Otherwise, subtraction is not done and a 0 is placed at the corresponding bit position of the quotient. Then we again shift the $2n$-bit number 1-bit left and repeat the process. This is equivalent to shifting the divisor to right and subtracting it from the high-order bits of the dividend.

13.5.1 Datapath

The block diagram of the binary divider is shown in Figure 13.14. The divisor is stored in register B, the dividend is stored in register Q, and remainder is formed in register A. Register A is initially set to 0. A parallel adder/subtractor adds the contents of register A and B or subtracts the content of B from the content of A and the result is stored in A. The C flip-flop is set to 0 or 1 corresponding to the quotient bit. The P counter is initially set to hold a binary number equal to the number of bits in the dividend. This counter is decremented after the formation of each remainder. When the content of the counter reaches 0, the remainder and the quotient are formed in A and Q, respectively, and the process stops. The control logic stays in an initial state until the start signal S becomes 1. The system then performs the division. Both the remainder in A and dividend in Q are shifted to the left. The most significant bit of Q is shifted into the least significant position of A; the value of the quotient bit in C is shifted into the least significant position of Q. After the shift left operation, one bit of the dividend is transferred into A while the remainder bits in A are shifted one position to the left. The most significant bit of register A, designated by A_{n-1}, holds the sign bit of the remainder that must be checked after subtraction to determine whether subtraction is possible. The control logic determines whether to add the divisor again to restore the value of the remainder if subtraction is not possible. The control logic also receives signal Z from a circuit that checks counter P for 0. A_{n-1} and Z are status inputs for the control unit. The start input S is an external control input. The outputs of the control logic activate the required operations in the registers.

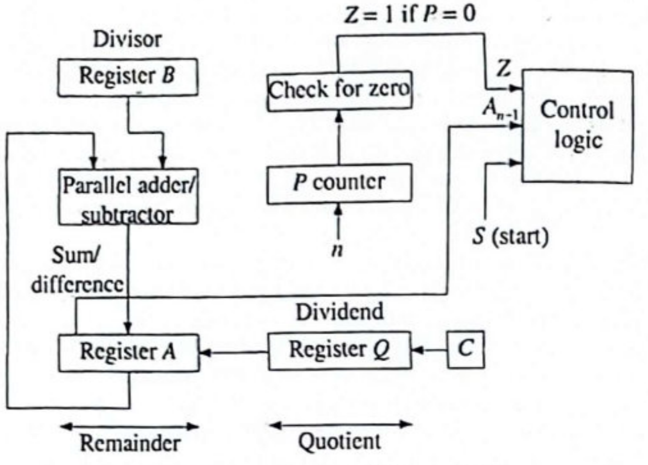


Figure 13.14 Block diagram of binary divider.

13.5.2 ASM Chart

The ASM chart for the binary divider is shown in Figure 13.15. Initially, the divisor is in register B and the dividend is in register Q. As long as the circuit is in the initial state and $S = 0$, no action occurs and the system remains in an initial state $T0$ (binary code 000). The division process starts when $S = 1$ and control goes to state $T1$ (binary code 001). Register A and flip-flop C are cleared to 0 and the sequence counter P is set to a binary number n, equal to the number of bits in the dividend. The system then goes to state $T2$ (binary code 010). In this state, the composite register AQC is shifted 1-bit left. Then the control goes to state $T3$ (binary code 011). In this state, the P counter is decremented by 1, B is subtracted from A, and the result is transferred to A. Then the remainder bit A_{n-1} is checked, and if it is equal to 0 (the remainder is positive and the subtraction is possible), then the control goes to state $T4$ (binary code 100), otherwise (the remainder is negative and subtraction is not possible), the control goes to state $T5$ (binary code 101). In state $T4$, the C flip-flop is set to 1, which is the required quotient bit, and the control goes to state $T6$ (binary code 110). In state $T5$, A and B are added to restore the remainder and the C flip-flop is set to 0, which is the required quotient bit. Then the control goes to state $T6$. In state $T6$, the composite register AQC is shifted 1-bit left to obtain the new remainder. Then the value in the P counter is checked. If the content of P is not 0, status bit Z is equal to 0 and the process is repeated from state $T3$ to obtain a new remainder and corresponding quotient bit. The control goes to state $T7$ (binary code 111) when the P counter reaches 0 and the control input Z is equal to 1. In state $T6$, the final quotient is formed in Q, but, in state $T7$, A is shifted 1-bit right to find the correct remainder.

The previous numerical example is repeated in Table 13.5 to clarify the division process. The procedure follows the steps outlined in the ASM chart of Figure 13.15.

The type of register needed for the data processing subsystem can be derived from the register operations listed in the ASM chart. Register A is a parallel-in parallel-out bi-directional shift register and has synchronous clear capability to reset the register to 0. It loads the result from the adder/subtractor in parallel and outputs the contents in parallel to add/subtract with B in the adder/subtractor. Initially its value is set to 0 by rest. Register Q is a parallel-in left-shift register. It loads the dividend in parallel. The counter P is binary down counter with a facility to parallel load a binary constant equal to the number of bits in the dividend. The C flip-flop must have a synchronous preset and clear. Register B is a parallel-in parallel-out register. It loads the divisor in parallel and outputs the contents in parallel to the adder/subtractor.

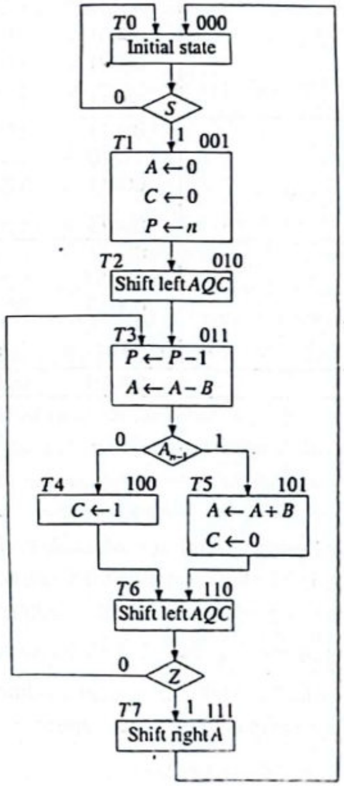


Figure 13.15 ASM chart for binary divider.

Table 13.5 Numerical example of binary division.

Divisor $B = 00101$	A	Q	C	P
Dividend in Q	XXXXX	11011	X	XXX
Division starts, $A = 0$, $C=0$, $P=101$	00000	11011	0	101
Shift left AQC	00001	10110	0	101
Decrement P	00001	10110	0	100
$A = A - B$	11100	10110	0	100
As $A_{n-1} = 1, A = A + B, C = 0$	00001	10110	0	100
Shift left AQC	00011	01100	0	100
Decrement P	00011	01100	0	011
$A = A - B$	11110	01100	0	011
As $A_{n-1} = 1, A = A + B, C = 0$	00011	01100	0	011
Shift left AQC	00110	11000	0	011
Decrement P	00110	11000	0	010
$A = A - B$	00001	11000	0	010
As $A_{n-1} = 0,\ C = 1$	00001	11000	1	010
Shift left AQC	00011	10001	1	010
Decrement P	00011	10001	1	001
$A = A - B$	11110	10001	1	001
As $A_{n-1} = 1, A = A + B, C = 0$	00011	10001	0	001
Shift left AQC	00111	00010	0	001
Decrement P	00111	00010	0	000
$A = A - B$	00010	00010	0	000
As $A_{n-1} = 0,\ C = 1$	00010	00010	1	000
Shift left AQC	00100	00101	1	000
Shift right A	00010	00101	1	000
Quotient $Q = 00101$ Remainder $A = 00010$				

13.5.3 Control Logic

The control logic is designed using multiplexer, sequence register, and decoder. As we have eight states, we need three bits to encode the states and we need three flip-flops $G2$, $G1$, and $G0$. Consequently we need three multiplexers MUX2, MUX1, and MUX0. The control inputs are S, Z, and A_{n-1}. We also need a 3×8 decoder to generate eight states $T0$ through $T7$. Multiplexer input conditions for binary divider is shown in Table 13.6. The corresponding control implementation is shown Figure 13.16.

Table 13.6 Multiplexer input conditions for binary divider.

Present state $G2G1G0$	Next state $G2G1G0$	Input condition	Inputs		
			MUX2	MUX1	MUX0
000	000	S'	0	0	S
000	001	S			
001	010		0	1	0
010	011		0	1	1
011	100	A'_{n-1}	1	0	A_{n-1}
011	101	A_{n-1}			
100	110		1	1	0
101	110		1	1	0
110	011	Z'	Z	1	1
110	111	Z			
111	000		0	0	0

Exercises

13.1 Explain in words the operations specified by the following register transfer notation:

(a) $R2 \leftarrow R2 + 1$, $R1 \leftarrow R2$

(b) $R3 \leftarrow R3 - 1$

(c) If ($T1 = 1$) then ($R0 \leftarrow R1$) else if ($T2 = 1$) then ($R0 \leftarrow R2$)

13.2 There are two control signals x and y. If $xy = 10$, register R is incremented by one and control goes from initial state to a second state. If $xy = 01$, register R is cleared to zero and control goes from the initial state to a third state. Otherwise, control stays in the initial state. Draw the portion of the ASM chart.

13.3 Draw the ASM chart for the following state transitions:

(a) If $x = 0$, control goes from state $T1$ to state $T2$; if $x = 1$, generate a conditional operation and go from $T1$ to $T2$.

(b) If $x = 1$, control goes from $T1$ to $T2$ and then to $T3$; if $x = 0$, control goes from $T1$ to $T3$.

(c) Start from state $T1$; then: if $xy = 00$, go to $T2$; if $xy = 01$, go to $T3$; if $xy = 10$, go to $T1$; otherwise, go to $T3$.

13.4 Show the eight exit paths in an ASM block emanating from the decision boxes that check the eight possible binary values of three control variables x, y, and z.

13.5 Construct an ASM chart for counting the number of 1s in a register.

13.6 Design the control for ASM chart of Figure 13.15 using sequence register and decoder.

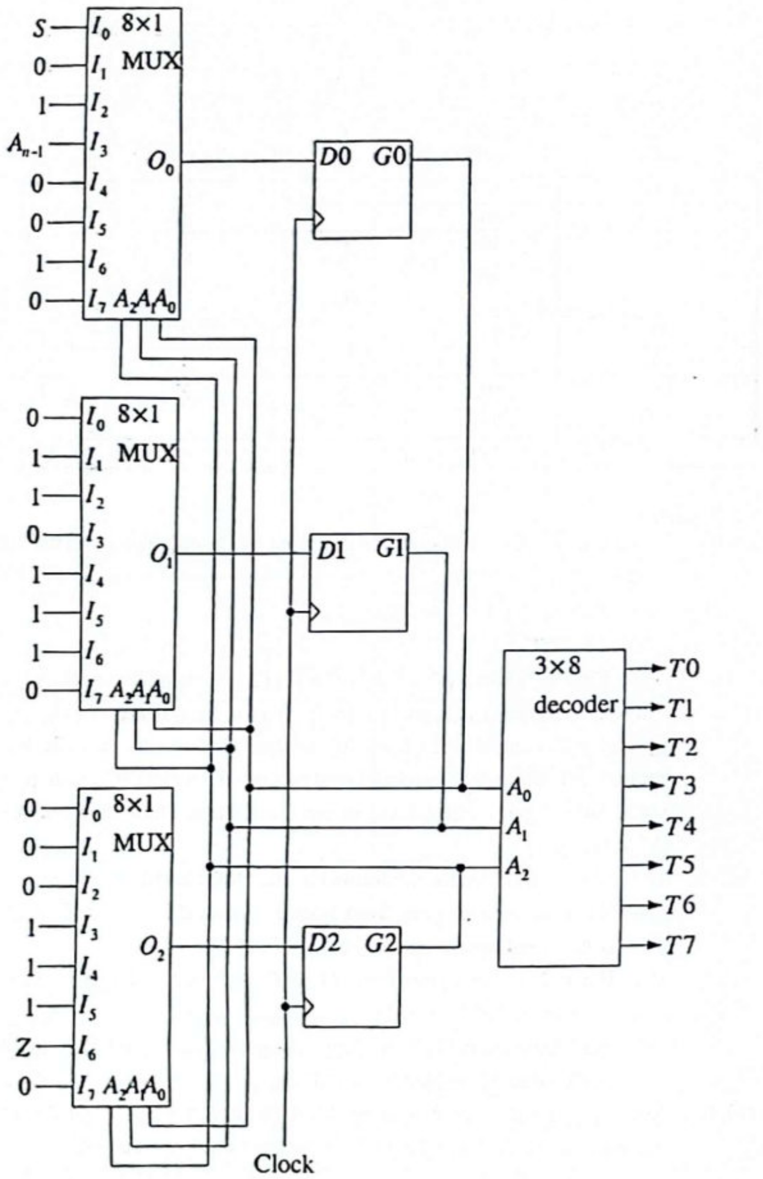


Figure 13.16 Logic diagram of control of binary divider using multiplexer, sequence register, and decoder.

13.7 Design the control for ASM chart of Figure 13.15 using one flip-flop per state.

13.8 The state diagram of a control unit is shown in Figure 13.17. Draw the equivalent ASM chart, leaving the state boxes empty.

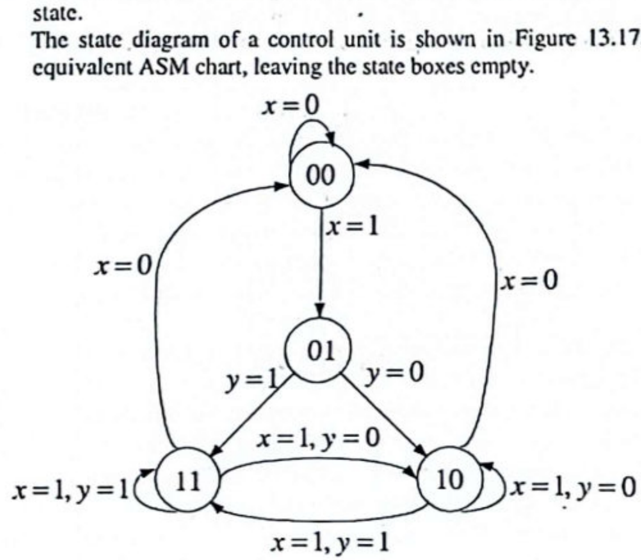


Figure 13.17 Control state diagram for exercise 13.8 and 13.9.

13.9 Design the control whose state diagram is shown in Figure 13.17.

13.10 Design a digital circuit that multiplies two binary numbers by the repeated addition method. For example, to multiply 6×3, the digital system evaluates the product by adding the multiplicand three times: 6 + 6 + 6 = 18. Let the multiplicand be in register *B*, the multiplier in register *A*, and the product in register *P*. An adder circuit adds the contents of *B* to *P*. A zero-detection circuit *Z* checks when *A* becomes 0 after each time it is decremented.

13.11 List the contents of registers *C*, *A*, *Q*, and *P* similar to Table 13.1 during the process of multiplying two numbers 11111 (multiplicand) and 10101 (multiplier).

13.12 List the contents of registers *A*, *Q*, *C*, and *P* similar to Table 13.5 during the process of dividing 11111 (dividend) by 10101 (divisor).

13.13 Determine the time it takes to process the multiplication operation in the binary multiplier designed in Section 13.4. Assume that the *Q* register has *n* bits and the clock cycle is *t* nanoseconds.

13.14 Determine the time it takes to process the division operation in the binary divider designed in Section 13.5. Assume that the *Q* register has *n* bits and the clock cycle is *t* nanoseconds.

13.15 Consider the ASM chart of Figure 13.18. The register operations are not specified, because we are interested only in designing the control logic.
(a) Draw the equivalent state diagram.

(b) Design the control with sequence register and decoder.
(c) Design the control with one flip-flop per state.
(d) Design the control with multiplexer, sequence register, and decoder.

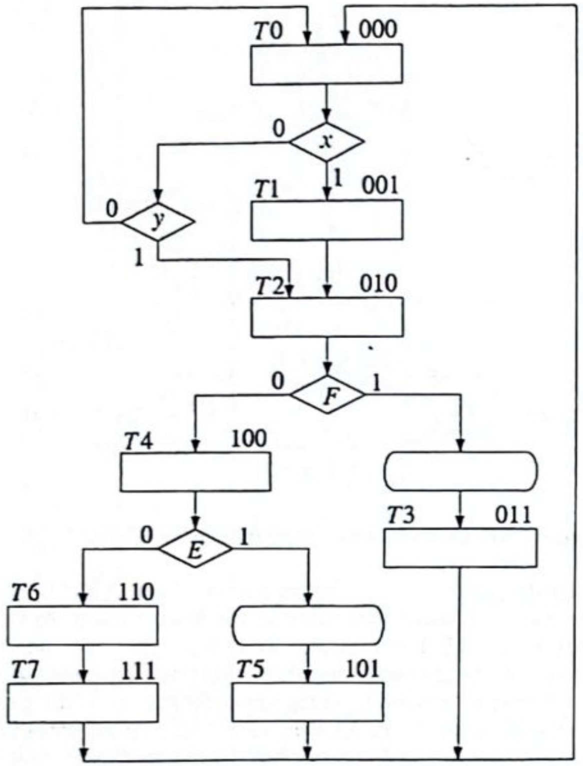


Figure 13.18 ASM chart for exercise 13.14.

Further Readings

1. M. Morris Mano and C R. Kime, *Logic and Computer Design Fundamentals*, 2^{nd} ed. Prentice Hall, 2000.
2. M. Morris Mano, *Digital Logic and Computer Design*, Prentice-Hall, Inc., 2000.
3. M. Morris Mano, *Digital Design*, 3^{rd} ed., Pearson Education, 2002.

CHAPTER 14
VERILOG HARDWARE DESCRIPTION LANGUAGE

14.1 Introduction

Digital systems are highly complex. At their detailed level, they may consist of millions of gates and flip-flops. From a more abstract viewpoint, these gates may be grouped into several functional components such as adders, subtractors, comparators, encoders, decoders, multiplexers, demultiplexers, registers, counters, control logic, etc. Hardware description languages have evolved to aid in the design of digital systems with large number of elements and wide range of electronic and logical abstractions.

The creative process of digital system design begins with a conceptual idea of a logical system to be built, a set of constraints that the final implementation must meet, and a set of primitive components from which to build the circuit. Design is an iterative process of either manually proposing or computer aided synthesizing alternative solutions and then testing them with respect to the given constraints. The design is typically divided into many smaller subsystems and each subsystem is further divided, until the whole design is specified in terms of known primitive components.

A hardware description language provides the digital circuit designer with a means of describing a digital system at a wide range of levels of abstractions, and, at the same time, provides access to computer-aided design tools to aid in the design process at these levels. The language supports the early conceptual stages of design with its behavioral constructs, and the later implementation stages with its structural constructs. The description may be simulated to determine correctness, and some synthesis tools exist for computer aided design.

In this chapter we will discuss structural and behavioral description and simulation of digital circuits using a very popular hardware description language - the Verilog. The most notable advantage of the Verilog language is that the syntax of the language is almost similar to that of the very popular programming language C. We will introduce 2001 standard of the Verilog language in a tutorial fashion, without going into discussing syntax and rules of the language in details.

14.2 Overview of Verilog Language

The Verilog language describes a digital system as a set of *modules*. A module represents a logical unit by specifying its internal logical structure, for instance describing the actual logic gates it is comprised of. Alternatively, a module may represent a logical unit by describing its behavior in a program-like manner focusing on what the module does rather than on its logical implementation. These modules are then interconnected with nets, allowing them to communicate.

14.2.1 Structural Description

The logic diagram of the *S* output of the full-adder circuit is shown in Figure 14.1. It is a three-input single-output circuit. A Verilog description of the circuit is shown in Figure 14.2. The description shows the basic definition of a module. Each module definition includes the keyword **module** followed by the module name (here the module name is *sumOfFA*) and is terminated by the **endmodule** statement. The second line of this definition specifies the names of *wires* used to transmit logic values among the submodules of this module. The third line declares the names of *registers* that will hold values.

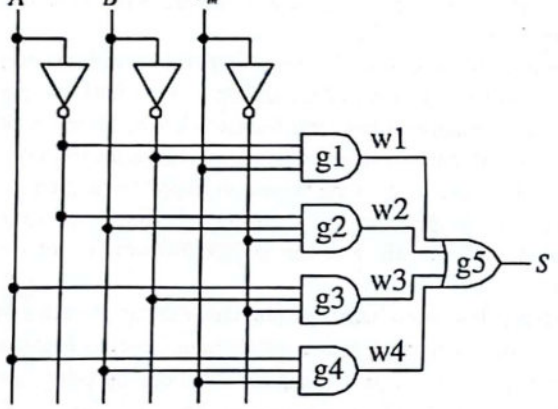


Figure 14.1 *S* output of the full-adder circuit.

```
module sumOfFA;
        wire    S, w1, w2, w3, w4;
        reg     A, B, Cin;

        and #1
                g1 (w1, ~A, ~B, Cin),
                g2 (w2, ~A, B, ~Cin),
                g3 (w3, A, ~B, ~Cin),
                g4 (w4, A, B, Cin);

        or #1
                g5 (S, w1, w2, w3, w4);
endmodule
```

Figure 14.2 Structural description of the *S* output of the full-adder circuit of Figure 14.1.

The fifth line, and its continuation onto lines 6 through 9, *instantiates* four AND gates, each having a delay of 1 time unit. The tenth line, and its continuation onto the eleventh line, instantiates one OR gate having a delay of 1 time unit. AND and OR gates are two of the predefined logic gate types in the language – the others are NAND, NOR, and XOR. These two statements specify that five gates, called *g1* through *g5*, exist in the circuit. The "#1" indicates that they each have a delay of 1 time unit. The labels in the parentheses indicate the wires and registers to which the gates are connected. The first label in the parentheses is the gate's output and the others are inputs. The NOT operator ("~") is used to specify that the complement of a value is connected to the input. The wire, register, and instance names are included in the logic diagram of Figure 14.1 to further clarify the correspondence between the logic diagram and its equivalent Verilog description.

The two fundamental data types of the Verilog language are *nets* and *registers*. Nets are used to model an electrical connection between structural entities such as gates. A *wire* is one type of net; others include wired-AND, wired-OR, and trireg connections.

In this example, AND and OR gates were used to build the *sumOfFA* module. This *sumOfFA* module, if it had input/output ports, could then be used in another module by instantiating it there, and so on.

As a final comment, it should be pointed out that the designation of *A*, *B*, and *Cin* as registers might seem anomalous. One would think that these would be inputs to module *sumOfFA*, and that the value *S* would be an output. These will be changed to inputs and output in a later example. But for now, we will keep the register definition as they will aid in simulation in the next subsection.

14.2.2 Simulating the Description

Figure 14.3 shows a more complete module definition for *sumOfFA* called *sumOfFASim*. The module includes statements that will provide stimulus to the AND gate instances, and statements that will monitor the changes in the output of the OR gate. In this module all possible eight combinations of 3-inputs *A*, *B*, and *Cin* are provided and the corresponding *S* outputs are monitored.

A simulator for a digital circuit is a program that executes the statements in Figure 14.3's *initial* statement (and as we will see in later examples, the *always* statement), and propagates changed values from the outputs of gates and registers to other gate and module inputs. A simulator is further characterized by its ability to keep track of *time*, causing the changed values to appear at some specified time in the future rather than immediately. These future changes are typically stored in a time-ordered event list. When the simulator has no further statement execution or value propagation to perform at the current time, it finds the next time-ordered event from the event list, updates time to that of the event, and executes the event. This event may or may not generate events at future times. This simulation loop continues until there are no more events to be simulated or the user halts the simulation by some other means.

Figure 14.3 differs from Figure 14.2 with the inclusion of the initial statement to drive the simulation. The simulator begins the simulation by starting the execution of the **initial** statement. The keywords **begin** and **end** bracket the individual statements that are part of the initial statement. The results of the simulation of this example are shown in Figure 14.4.

The first statement in the initial is a simulation command to monitor (and print) a set of values when any one of the values changes. In this case, the time is printed ($time requests that the current time be printed) and then the quoted string is printed with the values of *A*, *B*, *Cin*, and *S* substituted for the %b (for binary) printing control in the string. Between $time and the quoted string are several extra commas. One is needed to separate $time and the quoted string; the extras each introduce an extra space in the printout. When issued, the $monitor command prints the current values in the design, and will automatically print later when at least one of the values in its list changes. However, it will not print when only $time changes. As shown in Figure 14.4, they initially print as x. When the simulator starts, all values are unknown which is indicated by the x. The first value on the line is the time.

```
module sumOfFASim;
        wire    S, w1, w2, w3, w4;
        reg     A, B, Cin;
        and #1
                g1 (w1, ~A, ~B, Cin),
                g2 (w2, ~A, B, ~Cin),
                g3 (w3, A, ~B, ~Cin),
                g4 (w4, A, B, Cin);
        or #1
                g5 (S, w1, w2, w3, w4);
        initial // two slashes introduce a single line
                // comment
                begin
                        $monitor ($time,,,
                                "A=%b B=%b Cin=%b, S=%b",
                                A, B, Cin, S);
                        // waveform for simulating S output
                        #10 A=0; B=0; Cin=0;
                        #10 A=0; B=0; Cin=1;
                        #10 A=0; B=1; Cin=0;
                        #10 A=0; B=1; Cin=1;
                        #10 A=1; B=0; Cin=0;
                        #10 A=1; B=0; Cin=1;
                        #10 A=1; B=1; Cin=0;
                        #10 A=1; B=1; Cin=1;
                        #10 $finish;
                end
endmodule
```

Figure 14.3 *S* output of the full-adder to be simulated.

The initial statement continues by scheduling nine events to occur in the future. The statements:

```
#10 A=0; B=0; Cin=0;
```

specify that registers *A*, *B*, and *Cin* will each be loaded with 0 10 time units from the current time. The way to think about the execution of this line is that the simulator suspends the execution of the initial statement for 10 time units. The simulator sees no other action at the current time (time 0) and goes to the next event in the time-ordered event list, which happens to be this statement. Thus the simulator reactivates the initial statement. At that time (time 10) the initial statement is reactivated from where it suspended and the next statement is executed. At time 10, the registers *A*, *B*, and *Cin* are loaded with new value 0 and the $monitor command prints the second line of Figure 14.4. Then, the simulator continues executing on the next line where it sees the next #10. At this point the initial statement is suspended, waiting for ten more time units.

```
 0 A=x B=x Cin=x, S=x
10 A=0 B=0 Cin=0, S=x
12 A=0 B=0 Cin=0, S=0
20 A=0 B=0 Cin=1, S=0
22 A=0 B=0 Cin=1, S=1
30 A=0 B=1 Cin=0, S=1
40 A=0 B=1 Cin=1, S=1
42 A=0 B=1 Cin=1, S=0
50 A=1 B=0 Cin=0, S=0
52 A=1 B=0 Cin=0, S=1
60 A=1 B=0 Cin=1, S=1
62 A=1 B=0 Cin=1, S=0
70 A=1 B=1 Cin=0, S=0
80 A=1 B=1 Cin=1, S=0
82 A=1 B=1 Cin=1, S=1
```

Figure 14.4 Results of simulating module of Figure 14.3.

But at the current time (time 10), the changed values for *A*, *B*, and *Cin* are propagated. By propagation, we mean that every primitive gate that is connected to any of these is notified of the change. These gates may then schedule their outputs to change in the future. Because the gates in this example are defined to have a time delay of 1, their output changes will be propagated 1 time unit into the future (at time 11); the simulator schedules these values to be assigned and propagated then.

As mentioned above, the initial statement continued executing until it found the delay on the next line which specifies that in 10 more time units (i.e., at time 20), *Cin* will be loaded with a 1. The initial block is suspended and scheduled to wake up at time 20. The simulator looks for the next event in time, and it sees that four AND gates (*g1* through *g4*) are scheduled to change their output values at time 11 and propagate them to the OR gate, *g5*.

The result of propagating these four new values on wires *w1*, *w2*, *w3*, and *w4* is that gate *g5* will be scheduled to change its output value at time 12. Since there are no further events during the current time (time 11), the next event is taken from the event list at the next time (time 12). The change to *S* at time 12 will not cause any other gates to be evaluated because it is not connected to any other gates. At time 12, as the value of *S* is changed, the $monitor command prints the third line of Figure 14.4.

The simulation continues until the initial statement executes the $finish command. At time 20, *A*, *B*, and *Cin* are set to 0, 0, and 1, respectively and as the vale of *Cin* is changed, the $monitor command will print the forth line of Figure 14.4. This change of the register values will cause a change to *S* two time units later. Therefore, at time 22 the $monitor command will print the fifth line of Figure 14.4. At time 30, *A*, *B*, and *Cin* are set to 0, 1, and 0, respectively and as the vale of both *B* and *Cin* are changed, the $monitor command will print the sixth line of Figure 14.4. This change of the register values will not cause any change to *S*. Therefore, at time 32 the $monitor command will not print anything. Similarly, new values will be set to registers *A*, *B*, and *Cin* at time 40, 50, 60, 70, and 80 and new value of *S* will be generated (except for at time 72). Accordingly, the $monitor will print the seventh lines through the last line. At time 90, the $finish command stops the simulation program.

The simulator output in Figure 14.4 illustrates three of the four values that a bit may have in the simulator: 1 (TRUE), 0 (FALSE), and x (unknown). The fourth value, z, is used to model the high impedance outputs of tristate gates.

We now can see why *A*, *B*, and *Cin* were defined as registers for the examples of Figures 14.2 and 14.3. As the only "external" inputs to the AND gates, we needed a means of setting and holding their value during the simulation. Since wires do not hold values – they merely transmit values from outputs to inputs – a register was used to hold the input values.

So far we have seen the use of the two main data types in the language: nets and registers. Primitive gates are used to drive values onto nets; initial statements (and as we will see later, always statements) are used to make assignments to the registers.

14.2.3 Creating Ports for the Module

The example of Figure 14.2 had neither inputs nor outputs – a rather limiting situation that does not represent real modules nor help in developing a module hierarchy. The example of Figure 14.5 extends the notion of defining modules to include ones that have ports.

```
module sumOfFA
    (output   S,
    input     A, B, Cin);

    and #1
            g1 (w1, ~A, ~B, Cin),
            g2 (w2, ~A, B, ~Cin),
            g3 (w3, A, ~B, ~Cin),
            g4 (w4, A, B, Cin);
    or #1
            g5 (S, w1, w2, w3, w4);
endmodule
```

Figure 14.5 Adding ports to a module.

The first line is the opening module definition statement, using the module keyword and the module's name. On the second line, the opening parenthesis indicates that *ports* will be declared as part of this module. The ports are declared within the parenthesis to be *inputs*, *outputs*, or bidirectional *inouts*. Note that output(s) need not be first, as is the case with the primitive AND or OR gates. On the second line, this example declares *S* to be an output port. On the third line, three input ports named *A*, *B*, and *Cin* are declared. The closing parenthesis on the third line ends the declaration of the module ports. In contrast to Figure 14.2, *A*, *B*, and *Cin* are now wires that connect the input ports to the gates. The *sumOfFA* module of Figure 14.5 might be drawn in a logic diagram as shown in Figure 14.6. Note that the port names are shown on the inside of the module in the logic diagram. The port names are only known inside the module.

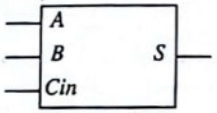


Figure 14.6 Logic diagram of the *sumOfFA* module of Figure 14.5.

This module may now be instantiated into other modules. The port list in the module definition establishes a contact between the internal workings of the module and its external usage. That is, there is one output and three inputs to connect to. No other connections within the module (say, wire *w1*) can be connected outside of this module. Indeed, the internal structure of the module is not known from the outside. Thus, once defined, the module is a blackbox that we can instantiate and connect into the design many times. But since we do not have to be bothered with the internal details of the module each time it is instantiated, we can control the descriptive complexity of the design.

In Figure 14.5, we no longer declare that *w1*, *w2*, *w3*, and *w4* are wires. Previously in Figure 14.2, we optionally chose to declare them as wires. Since gates only drive nets, these names, by default, are implicitly declared to be wires.

14.2.4 Creating a Testbench for a Module

When writing Verilog descriptions, it is often appropriate to organize the description using the *testbench* approach. In Verilog, a module is defined and, possibly, given the name *testBench*. Within this module are two other modules, one representing the circuit being designed, and the other representing the test generator and monitor. These are shown in Figure 14.7.

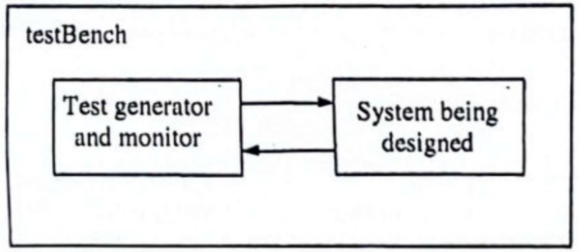


Figure 14.7 General view of a testbench module.

This is a clean way to separate the design's description and the means of testing it. The circuit being designed, shown on the right, can be simulated and monitored through its ports, and later the design might be synthesized using other CAD tools. The point is that the description being simulated and synthesized are the same. Further, all testing activity is encapsulated in the module on the left. If we include behavior to test a design within the design's module, then when we synthesize, we may need to remove this behavior, which is an error prone process. The *sumOfFA* module of Figure 14.3 showed a module where the design's description (the AND and OR gate instantiations) and the test behavior (the initial statement) were combined. Figure 14.8 shows this description rewritten using the testbench approach.

Module *testBench* instantiates two modules: the design module *sumOfFA* and the test module *testToSumOfFA*. When modules are instantiated, as shown on lines four and five, they are given names. The fourth line states that a module of type *sumOfFA* is instantiated in this module and given the name *sum*. The fifth line instantiates the *testToSumOfFA* module with name *test*. Now it is clear what the functionality of the system is and what its ports are. Further, it is clear how the system is going to be tested. The testbench approach separates this information, clarifying the description, and making it easier to follow on to other design tools such as logic synthesis. The connection of these two modules is illustrated in Figure 14.9.

Wires need to be declared when connecting modules together. Here, wires *c1* through *c4*, declared on line two of Figure 14.8, specify the interconnection. We can see that output register *A* in module *testToSumOfFA* is connected to an output of the module. The ports are specified and connected in order. In module *testToSumOfFA*, *A* is the second port (output port) and is connected to wire *c2*, which is also connected to second port *A* (input port) on module *sumOfFA*. Inside module *sumOfFA*, that port is connected to gates *g1*, *g2*, *g3*, and *g4*. Thus register *A* drives the inputs of *g1*, *g2*, *g3*, and *g4*. Simulating module *testBench* will produce the same results as simulating module *sumOfFASim* in Figure 14.3.

Within module *testToSumOfFA* we have declared *A*, *B*, and *Cin* to be registers. This is necessary because they are being assigned to in the initial block. Assignments in initial and always blocks must be to registers.

We might think that register *A* in module *testToSumOfFA* and input net *A* in module *sumOfFA* are the same because they are named the same. However, in Verilog, each module has its own name space; each *A* in this example is known only within the module in which it is declared. Thus the two *A*'s are names of distinct entities. In this example though, wire *c2* connects them making them electrically the same. Thus a change made to register *A* will propagate across wire *c2* to the *A* inputs of *g1*, *g2*, *g3*, and *g4*.

```
module testBench;
        wire       c1, c2, c3, c4;

        sumOfFA                     sum (c1, c2, c3, c4);
        testToSumOfFA               test (c1, c2, c3, c4);
endmodule
```

Figure 14.8 Using the testbench approach to description.

```
module sumOfFA
        (output  S,
        input    A, B, Cin);

        and #1
                 g1 (w1, ~A, ~B, Cin),
                 g2 (w2, ~A, B, ~Cin),
                 g3 (w3, A, ~B, ~Cin),
                 g4 (w4, A, B, Cin);
        or #1
                 g5 (S, w1, w2, w3, w4);
endmodule

module testToSumOfFA
        (input           S,
        output reg       A, B, Cin);

        initial
                begin
                        $monitor ($time,,,
                                "A=%b B=%b Cin=%b, S=%b",
                                A, B, Cin, S);
                        // waveform for simulating S output
                        #10 A=0; B=0; Cin=0;
                        #10 A=0; B=0; Cin=1;
                        #10 A=0; B=1; Cin=0;
                        #10 A=0; B=1; Cin=1;
                        #10 A=1; B=0; Cin=0;
                        #10 A=1; B=0; Cin=1;
                        #10 A=1; B=1; Cin=0;
                        #10 A=1; B=1; Cin=1;
                        #10 $finish;
                end
endmodule
```

Figure 14.8 Continued.

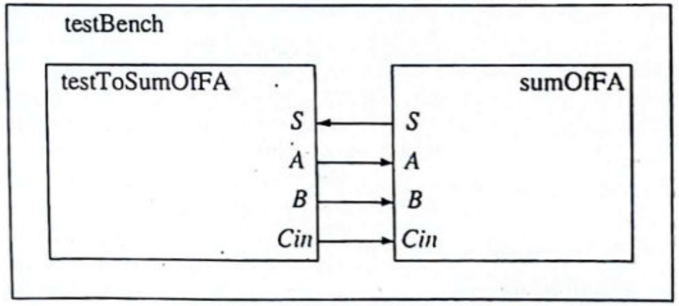


Figure 14.9 Interconnection of design and test module.

14.3 Behavioral Modeling of Combinational Circuits

So far we have highlighted the capabilities of the Verilog language of describing structure – module definitions, module instances, and their interconnections. The language also has the capability of describing a module's function behaviorally.

A *behavioral* model of a module is an abstraction of how the module works. The outputs of the module are described in relation to its inputs, but no effort is made to describe how the module is implemented in terms of structural logic gates. Behavioral models are described in a manner similar to a programming language.

14.3.1 Procedural Models

Figure 14.10 introduces the behavioral approach to modeling combinational logic. The functionality of the module is described in terms of procedural statements rather than with gate instantiations. The *always* statement is the basis for modeling behavior. The always statement, essentially a "while (TRUE)" statement, includes one or more *procedural* statements that are repeatedly executed. These procedural statements execute much like a software program: changing register values using the "=" procedural assignment, and executing loops and conditional expressions. Within the always statement, all assignment using "=" are made to entries declared as registers. This was also true of the initial statements seen earlier.

Figure 14.10 shows a behavioral model of *S* output of the full-adder. The port declarations are the same as before. We have also declared one register, S. This is the register we will make assignments to within the always statement, and it will also be the output of the purely combinational circuit. This always statement starts off with an event control "@" statement. The statement:

```
@(A, B, Cin) begin ... end
```

states that the simulator should suspend execution of this always block until a change occurs on one of the named entities. Thus, the value of each of *A*, *B*, and *Cin* is sampled when this statement executes. The simulator then waits for a change to occur on any of these named inputs. When a change occurs on any one or more of these, then execution will continue with the next statement – in this case what is contained in the begin ... end block.

```
module sumOfFABehavioral
        (output reg     S,
        input           A, B, Cin);
        always @(A, B, Cin) begin
                S=0;
                if (~A & ~B & Cin)
                        S=1;
                if (~A & B & ~Cin)
                        S=1;
                if (A & ~B & ~Cin)
                        S=1;
                if (A & B & Cin)
                        S=1;
        end
endmodule
```

Figure 14.10 A behavioral model of sumOfFA.

When a change occurs and execution continues, the assignment and if statements shown execute much like we would expect in a programming language. In this case, the statements describe how the output (*S*) is to be calculated from the inputs. The output is first set to 0. Then, if one of the input combinations producing functional output 1 is present in the inputs, *S* is set back to 1. The four if statements practically implement the four minterms of the *S* output of the full-adder (see Figure 7.4):

$$S = A' \cdot B' \cdot Cin + A' \cdot B \cdot Cin' + A \cdot B' \cdot Cin' + A \cdot B \cdot Cin$$

When the begin ... end block finishes, the always block restarts again, sampling the listed values (*A*, *B*, and *Cin*) and then waiting for a change on one or more of them. At this point, the simulator will propagate the final value of *S* to other parts of the design connected to it.

There are two features of the example of Figure 14.10. First, it describes the same functional behavior as the previous example of Figure 14.5, but there is no mention of the actual gate level implementation; the model is behavioral. Secondly, the fact that *s* is declared as a register might make us think that it is not a combinational circuit. But, let us consider the action of this module when only looking at its ports from the outside. We will quickly conclude that if there is any change on any of the inputs, the output will be re-evaluated based only on the module inputs. The previous value of *S* does not matter. This is a fundamental characteristic of a combinational circuit. From the outside of the module, it is clear that this has the behavior of a combinational circuit.

14.3.2 Rules for Synthesizing Combinational Circuits

Synthesis tools read a behavioral description of a circuit and automatically design a gate level structural version of the circuit. Thus, given example of Figure 14.10 as an input specification, a synthesis tool might produce the design specified in Figure 14.1; other implementations are possible too.

To use synthesis tools, we need to be very careful with how the description is written. The rules for synthesizing combinational circuits are briefly summarized below:

- All inputs to the combinational function are to be listed in the control event's sensitivity list (the comma-separated list of names). That way, if one of them changes, the output is re-evaluated. The need for this requirement stems from the definition of a purely combinational circuit. The output of a combinational circuit is a function of the current inputs; if one changes, the output should be re-evaluated.
- It is to be ensured that there is no way to execute the begin ... end loop without assigning a value to the combinational output (*s* in this example). That is, the output must be assigned a value at least once in every execution of the begin ... end loop. In Figure 14.10, line 6 (s=0;) assigns a value to *s* and satisfies this requirement. If the combinational output is not assigned a value during an execution of begin ... end loop, then the circuit needs to remember the previous value. Thus, the output is a function of the current inputs and the previous

output. This is a fundamental characteristics of a sequential circuit, not a combinational one.

Following these two rules will help us in writing behavioral description of combinational circuits that can be used equivalently for either simulation or synthesis.

14.3.3 Combinational Logic Description Using Continuous Assign

Figures 14.10 and 14.11 illustrate two different descriptions of the *S* output of the full-adder. Figure 14.11 uses the continuous assign statement for describing the logic function of the *S* output. Using a continuous assign statement is similar to specifying logic in Boolean algebra. The assign statement allows us to describe a combinational logic function without regard to its actual structural implementation – that is, there are no instantiated gates with wires and port connections. In a simulation of the circuit, the result of the logical expression on the right-hand side of the equal sign is evaluated anytime one of its values changes and the result drives the output *S*.

```
module sumOfFAAssign
        (output  S,
        input     A, B, Cin);

        assign    S = (~A & ~B & Cin) | (~A & B & ~Cin) |
                      (A & ~B & ~Cin) | (A & B & Cin);
endmodule
```

Figure 14.11 A synthesizable description of *S* output of the full-adder using continuous assign.

Continuous assign statements are often used for describing datapath elements. These modules tend to have one-line specifications as compared to many logic specifications. Figure 14.12 shows a description of a 4-bit binary parallel adder using continuous assign. The 4-bit inputs *A* and *B* and the 4-bit output *S* are declared as 4-bit *vectors*. The square brackets ("[]") construct declares the range of bit numbers that each wire has, the first number being the most-significant bit and the second being the least-significant bit. A single bit in a register or wire declared as a vector can be specified by *bit-select*. For example, the rightmost bit of the input *A* can be specified as *A[3]*. Registers and nets which are single-bit are said to be *scalar*. The sum generated on the right-hand side of the assign generates a result larger than output *S*. The concatenation operator "*{Cout, S}*" combines 1-bit *Cout* and 4-bit *S*; they are grouped (concatenated) together and treated as one 5-bit output. The concatenation operator specifies that the topmost bit (the carry out) will drive the *Cout* output and the rest of the bits will drive the *S* output.

```
module fourBitAdder
        (output          Cout,
        output [3:0]     S,
        input  [3:0]A, B,
        input            Cin);

        assign      {Cout, S} = A + B + Cin;
endmodule
```

Figure 14.12 Description of a four-bit parallel adder.

14.4 Procedural Statements to Specify Combinational Logic

In addition to using continuous assign statements and primitive gate instantiations to specify combinational logic, procedural statements may be used. Section 14.3.1 introduced this approach to specifying combinational logic. This section covers the topic in more detail.

14.4.1 The Basics

The basic form of a procedural description of combinational logic using if statement is shown in Figure 14.13. The example shows a 2×1 multiplexer described procedurally. It includes an always statement with an event statement containing all of the input variables to the combinational function. In this case, the address input *A* selects between passing data inputs *I0* or *I1* to output *O*. Even though *O* is defined to be a register, a synthesis tool will treat this module as a specification of combinational logic.

```
module twoByOneMux
     (output reg        O,
     input              A, I0, I1);

     always @(A, I0, I1)
          if (A == 0)
                    O = I0;
          else
                    O = I1;
endmodule
```

Figure 14.13 Procedural description of a 2×1 multiplexer using if statement.

The set of all registers, wires, and inputs used on the right-hand side of the procedural statements in the always block is defined to be the *input set* of the always block. In Figure 14.13, the input set contains *A*, *I0*, and *I1*. The list of names appearing in the event statement ("@") is defined to be the *sensitivity list* of an always block. In Figure 14.13, the sensitivity list contains *A*, *I0*, and *I1*. When describing combinational logic using procedural statements, every element of the always block's input set must appear in the sensitivity list of the event statement.

This follows from the very definition of combinational logic – *any* change of *any* input value may have an immediate effect on the resulting output. If an element of the input set is not in the sensitivity list, then it cannot have an immediate effect.

From Figure 14.13, we note that the combinational output *O* is assigned to in every branch of the if statement in the always block. A *control path* is defined to be a sequence of operations performed when executing an always loop. There may be many different control paths in an always block due to the fact that conditional statements (e. g., if and case) may be used. The output of the combinational function must be assigned in each and every one of the possible control paths. Thus, for every conceivable input change, the combinational output will be calculated anew.

The above example and discussion essentially outline the rules for specifying combinational hardware using procedural statements: *(i) the sensitivity list must be the input set and (ii) the combinational output(s) must be assigned to in every control path.*

The basic form of the "@" event statement is:

@ (sensitivity list) procedural statement;

If several procedural statements are needed to specify the combinational function, a begin ... end block must be used to group them into a compound statement.

An approach to organizing descriptions so that an assignment is always made is shown in Figure 14.14. This module has the same 2×1 multiplexer functionality as in Figure 14.13. However, here the output *O* is assigned to first. In a complex description, this approach ensures that the description will not be considered as a sequential circuit because of a forgotten output assignment.

```
module twoByOneMuxOutputFirst
        (output reg      O,
        input            A, I0, I1);

        always @(A, I0, I1)
                O = I0;
                if (A == 1)
                        O = I1;
endmodule
```

Figure 14.14 Procedural description of a 2×1 multiplexer with output assigned first.

14.4.2 Using Case Statements

Figure 14.15 illustrates using a case statement to specify a combinational function in a truth table form. Figure 14.15 specifies *S* output of the full-adder, which can be expressed as (see Figure 7.4)

$$S(A,B,Cin)=\sum(1,2,4,7)$$

The example illustrates and follows the rules for specifying combinational logic using procedural statements: all members of the always' input set are contained in the always' sensitivity list, and the combinational output is assigned to in every control path.

```
module sumOfFACase
        (output reg      S,
        input            A, B, Cin);

        always @(A, B, Cin)
                case ({A, B, Cin})
                        3'b000:         S=1'b0;
                        3'b001:         S=1'b1;
                        3'b010:         S=1'b1;
                        3'b011:         S=1'b0;
                        3'b100:         S=1'b1;
                        3'b101:         S=1'b0;
                        3'b110:         S=1'b0;
                        3'b111:         S=1'b1;
                endcase
endmodule
```

Figure 14.15 Description of *S* output of the full-adder using case statement.

The first line of the case specifies the concatenation of inputs *A*, *B*, and *Cin* as the means to select a case item to execute. The line following the case keyword specifies a numeric value followed by a colon. The number 3'b000 is the Verilog notation for a 3-bit number, specified here in binary as 000. The b indicates binary. The right-hand sides of the assignments of *S* need not be constants. Other expressions may be used that include other names.

When using a case statement, it is possible to incompletely specify the case. If there are n bits in the case's controlling expression, then a synthesis tool will know that there are 2^n possible control paths through the case. If not all of them are specified, then there will be a control path in which the output is not assigned to; it will be considered as sequential circuit. The default case item can be used to define the remaining unspecified case items. Thus Figure 14.15 could also be written as shown in Figure 14.16. Here, we explicitly list all of the 0s of the function using separate case items. If the input does not match one of these items, then by default *S* is assigned the value 1.

```
module sumOfFACaseWithDefault
        (output reg      S,
        input            A, B, Cin);

        always @(A, B, Cin)
                case ({A, B, Cin})
                        3'b000:  S=1'b0;
                        3'b011:  S=1'b0;
                        3'b101:  S=1'b0;
                        3'b110:  S=1'b0;
                        default: S=1'b1;
                endcase
endmodule
```

Figure 14.15 Description of *S* output of the full-adder using case statement with default.

14.4.3 Specifying Don't Care Situations

Logic synthesis tools make great use of logical don't care situations to optimize a logic circuit. Figure 14.16 illustrates specifying the *E3* output of BCD to Ex-3 code converter (see Figure 8.14) that contains don't cares. Often these can be specified in the default statement of a case. As shown, assigning the value x to the output is interpreted in this example as specifying input cases 4'b1010 through 4'b1111 to be don't cares.

```
module E3OfBCDToEx3CodeConverter
        (output reg        E3,
        input              B3, B2, B1, B0);

        always @( B3, B2, B1, B0)
                case ({B3, B2, B1, B0})
                        4'b0000: E3=1'b0;
                        4'b0001: E3=1'b0;
                        4'b0010: E3=1'b0;
                        4'b0011: E3=1'b0;
                        4'b0100: E3=1'b0;
                        4'b0101: E3=1'b1;
                        4'b0110: E3=1'b1;
                        4'b0111: E3=1'b1;
                        4'b1000: E3=1'b1;
                        4'b1001: E3=1'b1;
                        default:          E3=1'bx;
                endcase
endmodule
```

Figure 14.16 Description of E3 output of BCD to Ex-3 code converter including specification of don't care outputs.

14.4.4 Procedural Loop Constructs

If and case statements are not the only means to specify logic functions. The *for* loop in Verilog may be used to specify combinational logic. For loops allow for a repetitive specification as shown in Figure 14.17, which describes a 4-bit 2's complementer. In this example, each iteration of the for loop specifies a different logic element indexed by the loop variable *i*. The for loop of the example assigns the complement of the *i*th bit of *din* to the *i*th bit of *dout*. After the execution of the for loop *dout* will contain the 1's complement of *din*. Then the last assignment adds a 1 to *dout* to make it 2's complement of *din*. The @(*) is shorthand for "all the signals on the right-hand side of the statements or in conditional expressions", which is equivalent to listing all the names of the input set in the sensitivity list of the event statement @. This shorthand frees the designer from the burden of listing all names of the input set in the sensitivity list.

```
module twosComplementer
            (output reg      [1:4]      dout,
            input            [1:4]      din);

            reg              [1:4]      i;

            always @(*) begin
                    for (i=1; i<= 4; i=i+1)
                            dout[i]=~din[i];
                    dout=dout+1;
            end
endmodule
```

Figure 14.17 Description of a 4-bit 2's complementer using for loop.

14. 5 Procedural Modeling of Clocked Sequential Circuits

The model of a *clocked sequential circuit,* also called *finite state machine,* of Figure 9.1 can be redrawn as in Figure 14.18.The output of the current state flip-flops is the current state of the machine. Their input is the next state of the machine which the registers will load after the clock edge. The next state is a combinational function of the current state and the inputs. The outputs are a combinational function of the current state and the inputs. This structuring appears in the Verilog description. The next state and output combinational functions will be described behaviorally in a single always block following the rules of designing combinational circuit. The state registers will be described in a separate always block following a different set of rules.

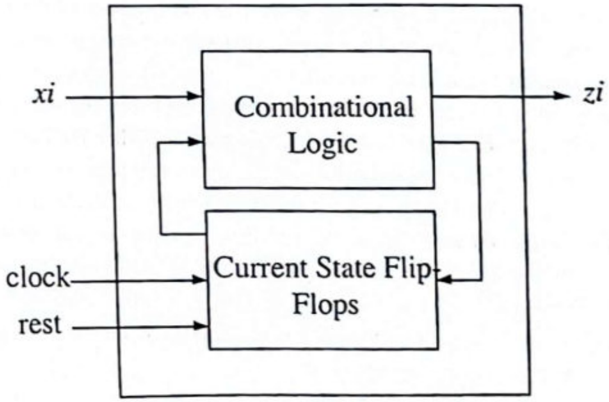


Figure 14.18 Model of a finite state machine.

14.5.1 Modeling Finite State Machines

An example transition diagram of a finite state machine is shown in Figure 14.19 and its behavioral description is shown in Figure 14.20. We have named the output *out*, the input *in*, and have also provided ports for *clock* and *reset*. The output *out* has also been declared to be a register. The current state and the next state of the machine have been named *currentState* and *nextState*, respectively. They have been declared to be 2-bit register vectors. The out, currentState, and nextState are assigned to in the combinational always block that implements the output function and next state.

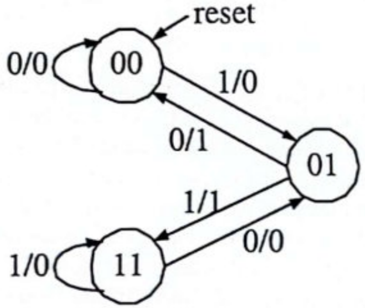


Figure 14.19 An example transition diagram.

The first always block describes the combinational behavior of the output and next state logic. The sensitivity list indicates that when a change occurs to *in* or *currentState*, then the begin ... end statement is executed. The statement in the begin ... end specify the new values of the combinational outputs *out* and *nextState*. From the transition diagram of Figure 14.19, we see that *out* is 1 when the *currentState* is 01, otherwise it is 0. Therefore, *out* is first assigned to 0 and then it is assigned to 1 if *currentState* is 1 (the binary constant 2'b01 is written as decimal number 1). When *currentState* is 00 and *in* is 1, then *nextState* is 01. When *currentState* is 01 and *in* is 1, then *nextState* is 11. When *currentState* is 11 and *in* is 1, then *nextState* is 11. When *currentState* is 11 and *in* is 0, then *nextState* is 01. Otherwise *nextState* is 00. Therefore, *nextState* is first assigned to 0 (2'b00) and then depending on the condition it is assigned to appropriate value.

```
module fsm
        (output reg         out,
        input               in, clock, reset);

        reg      [1:0]      currentState, nextState;

        always @(in, currentState) begin
        // the combinational portion
                out = 0;                // determining output
                    if (currentState == 1)
                            out = 1;

                  nextState = 0; // determining next state
                    if (currentState == 0)
                            if (in) nextState = 1;
                    if (currentState == 1)
                            if (in) nextState = 3;
                    if (currentState == 3) begin
                            if (in)
                                    nextState = 3;
                            else
                                    nextState = 1;
                    end
        end

        always @(posedge clock, negedge reset) begin
        // the sequential portion
                    if (~reset)
                            currentState <= 0;
                    else
                            currentState <= nextState;
        end
endmodule
```

Figure 14.20 Synthesizable description of finite state machine of Figure 14.19.

The second always block specifies the sequential portion of the finite state machine. In earlier initial and always blocks, we used regular assignment "=". In this always block, we used *non-blocking* assignment "<=". The non-blocking assignment is used to synchronize assignment statements so that they all appear to happen at once – concurrently. The non-blocking assignment is used in initial and always statements with edge specifications (i. e., posedge or negedge). When the specified edge occurs, then the new values are loaded concurrently in all assignments that were waiting for the signal's edge. In contrast to the regular assignment ("="), the right-hand sides of all non-blocking assignments waiting for the signal's edge are evaluated first, and then the left-hand sides are assigned (updated). When all the flip-flops in a large digital system are clocked from the same clock edge, the outputs of the flip-flops are updated concurrently. The non-blocking assignment models this behavior.

The sensitivity list in the always block waits for one of two events to occur: either a positive edge on *clock* or a negative edge on *reset*. When one or both of these occur, the begin ... end block is executed. Let us assume that a negative edge on *reset* occurs. As the begin ... end block begins executing, *reset* will be 0 and thus *currentState* will be set to 0. As long as *reset* remains 0, even a positive edge on *clock* will only result in *currentState* being set to 0. This is the action of an asynchronous reset signal that overrides the clock on a flip-flop.

Now let us consider the situation where *reset* is 1 and there is a positive edge on *clock*; the begin ... end loop is executed but this time the else clause is taken. The assignment

```
currentState <= nextState;
```

loads *currentState* with the *nextState*. These statements model the positive edge-triggered behavior of a 2-bit register made up of D flip-flops.

Let us now see how the whole finite state machine model works. Let us assume that *currentState* is 0, *reset* is 1, *clock* is 0, and *in* is 1. Then the two combinational outputs are: *out* is 0, and *nextState* is 1. When the positive edge of the clock occurs, the second always block will execute and assign the value of *nextState* to *currentState* and then wait again for the next positive edge of *clock* or negative edge of *reset*. Since *currentState* just changed to 1, the first always block will execute and calculate a new value for *out* and *nextState*. *out* will become 1, and *nextState* will become 3 if *in* remains 1. If *in* becomes 0, the first always block will execute again, recalculating *out* and *nextState* independent of the clock edge; *nextState* will become 0, and *out* will become 0.

14.5.2 Rules for Synthesizing Sequential Systems

There are rules for synthesizing sequential systems. The sequential portion of Figure 14.20 is the second always block. The rules are:

- The sensitivity list of the always block includes only the edges for the clock, reset and preset conditions. These are the only inputs that can cause a state change.
- Inside the always block, the reset and preset conditions are specified first. If a negative edge on reset was specified, then the if statement should be "if (~reset) ...". If a positive edge on reset was specified, then the if statement should be "if (reset) ...".
- A condition on the clock is not specified within the begin ... end block. The assignment in the last else is assumed by the synthesis tool to be the next state.
- Any register assigned to in the sequential always block will be implemented using flip-flops in the resulting synthesized circuit. Thus we cannot describe purely combinational logic in the same always block where we describe sequential logic. We can write a combinational expression, but the result of that expression will be evaluated at a clock edge and loaded into a register.
- Non-blocking assignment ("<=") are the assignment operator of choice when specifying the edge-sensitive behavior of a circuit. The "<=" states that all the transfers in the whole system that are specified to occur on the edge in the sensitivity list should occur concurrently. Although description using the regular "=" will synthesize properly, they may not simulate properly. Since both simulation and synthesis are generally of importance, we use "<=" in this situation.

14.6 Explicit Style of Describing Finite State Machines

In explicit style of describing finite state machines, a case statement is used to specify the actions in each of the machine's states and the transition between states. An example state diagram is shown in Figure 14.21, where six states and their state transitions are shown with one input and three output bits specified. Figure 14.22 shows the Verilog description of this finite state machine.

The first always statement is a description of the combinational output (*out*) and next state (*nextState*) functions. The input set for these functions contains the input *i* and the register *currentState*. Any change on either of these will cause the always statement to be re-evaluated. The single statement within the always is a case statement indicating the actions to be performed in each state. The controlling expression for the case is the state variable (*currentState*). Thus, depending on what state the machine is in, only the specified actions occur. In each case item, the two combinational functions being computed (*out* and *nextState*) are assigned to using conditional operator. In addition, a default case is listed representing the remaining unassigned states. The default sends the machine to state *A* which is equivalent to a reset. By arbitrary choice, *out* is set to don't care in the unassigned states.

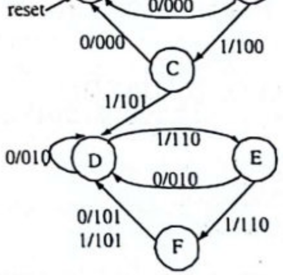


Figure 14.21 State diagram of an example finite state machine.

This always statement will result in combinational logic, because the sensitivity list contains all of the input set, there are no edge specifiers in the sensitivity list, and for every control path, both of the combinational outputs have been assigned to.

In the second always statement, *reset* is asserted low and will cause the machine to go into state *A*. If *reset* is not asserted, then the normal action of the always will be to load *currentState* with the value of *nextState*, changing the state of the finite state machine on the positive edge of *clock*.

The localparam statement specifies the state assignment for the system. Since these are treated as constants, they cannot be directly overridden by instantiation.

```
module fsm
        (input                  i, clock, reset,
        output   reg [2:0] out);
        reg      [2:0]     currentState, nextState;

        // The state labels and their assignments
        localparam          [2:0]
                A = 3'b000, B = 3'b001, C = 3'b010,
                D = 3'b011, E = 3'b100, F = 3'b101;

        always @(*)
                case (currentState)
                A:      begin
                        nextState = (i == 0) ? A : B;
                        out = (i == 0) ? 3'b000 : 3'b100;
                        end
                B:      begin
                        nextState = (i == 0) ? A : C;
                        out = (i == 0) ? 3'b000 : 3'b100;
                        end
                C:      begin
                        nextState = (i == 0) ? A : D;
                        out = (i == 0) 3'b000 : 3'b101;
                        end
                D:      begin
                        nextState = (i == 0) : D : E;
                        out = (i == 0) ? 3'b010 : 3'b110;
                        end
                E:      begin
                        nextState = (i == 0) ? D : F;
                        out = (i == 0) ? 3'b010 : 3'b110;
                        end
                F:      begin
                        nextState = D;
                        out = 3'b101;
                        end
                        default: begin  // undefined states. Go to state A
                        nextState = A;
                        out = 3'bxxx;
                        end
                endcase
```

Figure 14.22 Verilog description of the finite state machine of Figure 14.21.

```
        always @ (posedge clock, negedge reset)  // The state register
                if (~reset)
                        currentState <= A;  // the reset state
                else
                        currentState <= nextState;
endmodule
```

Figure 14.22 Continued.

14.7 Hierarchical Description of a Large System

Figure 14.23 shows pictorially a BCD counting system. The system consists of a board module which contains a clock module, a BCD counter, and a BCD to 7-segment display decoder. In this section we will detail each of the modules and their interconnection.

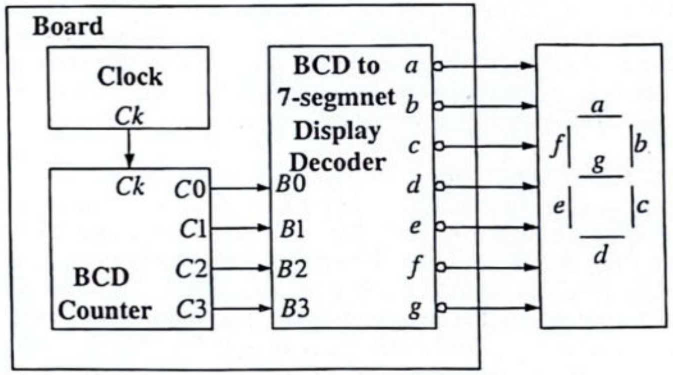


Figure 14.23 A BCD counting system.

14.7.1 The Clock Generator

The BCD counter of the system needs a clock to drive it. Figure 14.24 shows behavioral description of a counter and Figure 14.25 shows the waveform generated from simulating the description. This module is added here to simulate the counting system and can not be synthesized by a synthesis tool.

```
module clockGenerator
        (output  reg       clock);

        initial
                #5 clock = 1;

        always
                #50 clock = ~clock;
endmodule
```

Figure 14.24 Description of a clock generator.

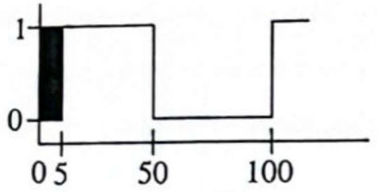


Figure 14.25 Wave shape of the generated clock.

The *clockGenerator* module has an internal register (*clock*) which is also the output of the module. At the start of a simulation, the output has the value **x** as illustrated by the dark area in Figure 14.25. We choose to initialize the *clock* after 5 time units have passed. The always statement states that after 50 time units *clock* will be loaded with its complement. Since an always statement executes until the simulation is stopped, after the first 50 time units have passed, the always statement will be scheduled to execute and change *clock*'s value in another 50 time units; i. e., this time at time 100. Because *clock* will change value every 50 time units, we have created a clock with a period of 100 time units.

14.7.2 The BCD Counter

The BCD counter definition is shown in Figure 14.26. The counter has two ports: the 4-bit counter register *C*, and a clock to increment the counter. The description declares that the internal register *C* and its output port are 4-bit vectors and provides an initial simulation value for *C* (0); when simulation begins, *C* will be set to the constant value specified to the right of the "=". The counter is modeled behaviorally using an always block. The module waits for a positive edge on *clock*. When that occurs, *C* is incremented. If the value of *C* is 10, then it is set to 0 to make it a BCD counter. Since the generation of the new counter value occurs on an edge of the clock signal, the non-blocking assignment operator ("<=") is used.

```
module BCDCounter
      (output    reg      [3:0]     C = 0,
       input     clock);

       always @ (posedge clock) begin
                     C <= C + 1;
                     if (C == 10)
                         C <= 0;
       end
endmodule
```

Figure 14.26 The BCD counter.

If *C* had not been initialized, its bits would always be unknown. When the simulation tries to increment a register whose bits are unknown (*x*), the result is unknown. Thus, for simulation *C* must be initialized. However, the synthesis tools will ignore this initialization.

14.7.3 The BCD to 7-Segmnet Display Decoder

The truth table of a BCD to 7-segment display decoder with active-LOW output is shown in Figure 8.19, which is reproduced here in Figure 14.27. The logic circuit of the BCD to 7-segmnet display decoder is shown in Figure 8.21 and is reproduced here in Figure 14.28. In Figure 14.28, the gates are labeled *g*1 through *g*20 and the wires are labeled *w*1 through *w*13. The Verilog description of the logic circuit of Figures 14.28 is shown in Figure 14.29. The module *BCDToSevenSeg* has seven output ports (*a*, *b*, *c*, *d*, *e*, *f*, and *g*) and four input ports (*B3*, *B2*, *B1*, and *B0*). The module is described structurally.

14.7.4 The Top-level Module – Tying the Whole Circuit Together

We have defined the basic modules to be used in our counting system in Figures 14.24, 14.26, and 14.29. The only thing remains is the tying of these modules together to complete the design shown in Figure 14.23. The module of Figure 14.30 ties together the module definitions of Figures 14.24, 14.26, and 14.29 by defining another module called *board* that instantiates and interconnects these modules.

BCD inputs				7-segment outputs						
B_3	B_2	B_1	B_0	a	b	c	d	e	f	g
0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	1	0	0	1	1	1	1
0	0	1	0	0	0	1	0	0	1	0
0	0	1	1	0	0	0	0	1	1	0
0	1	0	0	1	0	0	1	1	0	0
0	1	0	1	0	1	0	0	1	0	0
0	1	1	0	0	1	0	0	0	0	0
0	1	1	1	0	0	0	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	1	0	0
1	0	1	0	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1

Figure 14.27 Truth table of a BCD to 7-segmnet display decoder with active-LOW output.

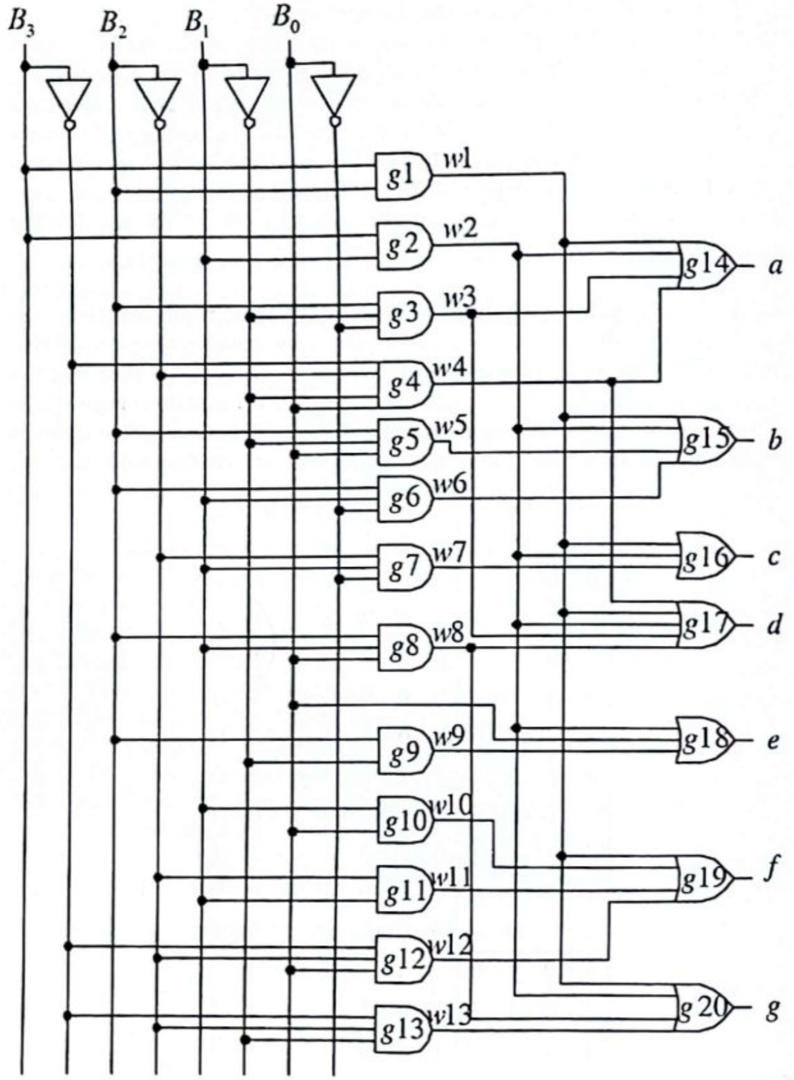

B_3
B_2
B_1
B_0
g1
g2
g3
g4
g5
g6
g7
g8
g9
g10
g11
g12
g13
g14
g15
g16
g17
g18
g19
g20
w1
w2
w3
w4
w5
w6
w7
w8
w9
w10
w11
w12
w13
a
b
c
d
e
f
g

Figure 14.28 Logic circuit for BCD to 7-segmnet display decoder with active-LOW outputs.

```
module BCDToSevenSeg
    (output  a, b, c, d, e, f, g,
    input    B3, B2, B1, B0);

    and #1
              g1 (w1, B3, B2),
              g2 (w2, B3, B1),
              g3 (w3, B2, ~B1, ~B0),
              g4 (w4, ~B3, ~B2, ~B1, B0),
              g5 (w5, B2, ~B1, B0),
              g6 (W6, B2, B1, ~B0),
              g7 (w7, ~B2, B1, ~B0),
              g8 (w8, B2, B1, B0),
              g9 (w9, B2, ~B1),
              g10 (w10, B1, B0),
              g11 (w11, ~B2, B1),
              g12 (w12, ~B3, ~B2, B0),
              g13 (w13, B3, ~B2, ~B1);

    or #1
               g14 (a, w1, w2, w3, w4),
               g15 (b, w1, w2, w5, w6),
               g16 (c, w1, w2, w7),
               g17 (d, w1, w2, w3, w4, w8),
               g18 (e, w2, B0, w9),
               g19 (f, w1, w10, w11, w12),
               g20 (g, w1, w2, w8, w13);
endmodule
```

Figure 14.29 Description of BCD to 7-segmnet display decoder.

```
module board;
      wire       [3:0]      count;
      wire                  clock, a, b, c, d, e, f, g;

      clockGenerator               clockGen (clock);
      BCDCounter                   counter  (count, clock);
      BCDToSevenSeg                display  (a, b, c, d, e, f, g, count[3],
                                            count[2], count[1], count[0]);

      initial
            $monitor ($time,,, "count=%d, abcdefg=
            %b%b%b%b%b%b%b", count, a, b, c, d, e, f, g);
endmodule
```

Figure 14.30 The top-level module of the counting system.

The module declaration of the *BCDCounter* of Figure 14.26 shows two ports, *C* and *clock*. *C* is a 4-bit output and *clock* is a 1-bit input. In Figure 14.30, the output of the *BCDCounter* module is connected to the input of the *BCDToSevenSeg* module. However, in Figure 14.29, *BCDToSevenSeg* module is defined to have seven 1-bit output ports and four 1-bit input ports. In the board module of Figure 14.30, we define a 4-bit wire *count* that connects to *BCDCounter*. When we connect it to *BCDToSevenSeg* we need to connect each bit (*B3*, *B2*, *B1*, *B0*) individually. This we do with a bit-select of the *count* wire, specifying the appropriate bit for each connection – *count[3]* to *B3*, *count[2]* to *B2*, *count[1]* to *B1*, and *count[0]* to *B0*. Alternatively, we could have declared four scalar (single-bit) wires and then concatenated them together when connecting to the *BCDCounter* module. In that case the *board* module description would have been as shown in Figure 14.31. Here we define scalar wires *w3*, *w2*, *w1*, *w0* and we connect them to ports *B3*, *B2*, *B1*, *B0* of *BCDToSevenSeg*. However, module *BCDCounter* expects a 4-bit vector to connect to its *C* port. The concatenation operator "{*w3*, *w2*, *w1*, *w0*}" combines *w3*, *w2*, *w1*, *w0*.

```
module board;
      wire          w3, w2, w1, w0, clock, a, b, c, d, e, f, g;

      clockGenerator       clockGen (clock);
      BCDCounter           counter          ({w3, w2, w1, w0}, clock);
      BCDToSevenSeg   display (a, b, c, d, e, f, g, w3, w2, w1, w0);
           initial
                 $monitor ($time,,, "count=%d, abcdefg=
                 %b%b%b%b%b%b%b", count, a, b, c, d, e, f, g);
endmodule
```

Figure 14.31 An alternate top-level module of the counting system.

If the module definitions of Figures 14.24, 14.26, 14.29, and 14.30 are compiled together, they form a complete description that can be simulated. The simulation trace from simulating these combined modules for 802 time units is shown in Figure 14.32.

```
0 count=0, abcdefg=xxxxxxx
2 count=0, abcdefg=0000001
5 count=1, abcdefg=0000001
7 count=1, abcdefg=1001111
100 count=2, abcdefg=1001111
102 count=2, abcdefg=0010010
200 count=3, abcdefg=0010010
202 count=3, abcdefg=0000110
300 count=4, abcdefg=0000110
302 count=4, abcdefg=1001100
400 count=5, abcdefg=1001100
402 count=5, abcdefg=0100100
500 count=6, abcdefg=0100100
502 count=6, abcdefg=0100000
600 count=7, abcdefg=0100000
602 count=7, abcdefg=0001111
700 count=8, abcdefg=0001111
702 count=8, abcdefg=0000000
800 count=9, abcdefg=0000000
802 count=9, abcdefg=0000100
```

Figure 14.32 Simulation trace of Figures 14.24, 14.26, 14.29, and 14.30.

Initially, all values in the system at time 0 are unknown. Then, the initial and always blocks, as well as the initializations in the declarations (e. g., the initialization of *C*), are enabled to run; they begin running in an arbitrary order. The initial statement in *clockGenerator* begins by delaying for #5 and the always begins by delaying for #50. The always in *BCDCounter* begins by waiting for a positive edge on the clock. The initialization in the declaration sets *C* to 0. The gate primitives in *BCDToSevenSeg* wait for a change on their inputs. The initial statement in *board* also runs. We can see that the initialization in *BCDCounter* runs first, setting *C* to 0. Then the initial in *board* runs, executing the $monitor statement and printing the first line in Figure 14.32.

Given the *C* (count) is set to 0 at time 0, two time units later outputs *a* through *g* change their values. At time 5, *clock* changes from **x** to 1. In Verilog, this is interpreted as a positive edge, which changes *C* (count) to 1. Two time units later, at time 7, outputs *a* through *g* change their values. At time 50, *clock* changes to 0. However, this is not shown in our simulation because we were not monitoring the changes in the *clock*. At time 100, *clock* changes to 1, creating a positive edge on *clock* and incrementing *C* (count). *C* changes to 2 and outputs change their values appropriately two time units later. The simulation continues as shown.

14.8 Tying Behavioral and Structural Models Together

In Figure 14.30, we connected together modules that were defined differently. Modules of Figures 14.24 and 14.26 were defined behaviorally and module of Figure 14.29 was defined structurally. This is a powerful aspect of the language because it allows us to model parts of a system at a detailed level (i. e., the structural models) and other parts at a less detailed level (the behavioral models). At the start of a design project, most of the system will be at the behavioral level. Then parts will be detailed into structural models. The final simulation could then be with all modules defined at the gate level for accurate timing and functional simulation. Thus the language aids in the complete design process, allowing a design to evolve from behavioral through to structural in evolutionary steps.

Indeed we can use both structural description and behavioral description within a single module. The module of Figure 14.29 can be rewritten as shown in Figure 14.33. Here the first level gates are described structurally and the outputs *a* through *g* are described behaviorally which ORs the outputs of the appropriate AND gates.

Exercises

14.1 Rewrite the module of Figure 14.2 with continuous assignment statement.

14.2 Write description of a 2-bit full-adder including carry-in and carry-out ports using gate-level models.

14.3 Write description of a 2-bit full-adder including carry-in and carry-out ports using continuous assignment statement.

14.4 Write description of a 2-bit full-adder including carry-in and carry-out ports using combinational always.

14.5 Change the clock generator of Figure 14.24 such that the clock period remains the same but that the low pulse width is 40 and high pulse width is 60.

14.6 Write a two-phase clock generator, where phase two should be offset from phase one by one quarter of a cycle.

```
module BCDToSevenSeg
        (output reg       a, b, c, d, e, f, g,
        input             B3, B2, B1, B0);

        and #1
                g1 (w1, B3, B2),
                g2 (w2, B3, B1),
                g3 (w3, B2, ~B1, ~B0),
                g4 (w4, ~B3, ~B2, ~B1, B0),
                g5 (w5, B2, ~B1, B0),
                g6 (W6, B2, B1, ~B0),
                g7 (w7, ~B2, B1, ~B0),
                g8 (w8, B2, B1, B0),
                g9 (w9, B2, ~B1),
                g10 (w10, B1, B0),
                g11 (w11, ~B2, B1),
                g12 (w12, ~B3, ~B2, B0),
                g13 (w13, B3, ~B2, ~B1);

        always @ (*) begin
                a = w1 | w2 | w3 | w4;
                b = w1 | w2 | w5 | w6;
                c = w1 | w2 | w7;
                d = w1 | w2 | w3 | w4 | w8;
                e = w2 | B0 | w9;
                f = w1 | w10 | w11 | w12;
                g = w1 | w2 | w8 | w13;
        end
endmodule
```

Figure 14.33 Description of BCD to 7-segmnet display decoder.

14.7 A serial adder can be described as below: *a* and *b* are two bits to add and *sum* is the result. The bits will come in to the serial adder low order first. The carry into the low-order bits is assumed to be 0. The inputs are all valid before the negative-edge of the clock. The serial adder has a *start* input. If *start* is 1, then the low-order *a* and *b* bits are added. Just after the negative-edge of the clock, *sum* will be updated with its new value based on the values of *a*, *b*, and the carry from the previous bits just before the clock edge. Write a behavioral description of the serial adder. Create a test module to demonstrate the serial adder's correctness.

14.8 Write the Verilog module for the K-map of Figure 14.34. Also write a test module to verify the correctness of the description. The output of the execution of the simulation module must relate easily to a truth table description of the function and the minterms represented on the K-map. Use a for loop to generate the test values in the test bench.

$AB \backslash CD$	00	01	11	10
00	0	1	0	0
01	0	1	1	1
11	1	1	1	0
10	0	0	1	0

Figure 14.34 K-map for exercise 14.8.

14.9 Draw the circuit described by the following Verilog module. Completely label all signals and gates, including propagation delays.

```
module exercise9
    (output       f,
    input         a, b, c);

    nand #10
        g1 (f1, a, b),
        g2 (f, f1, c);
endmodule
```

14.10 Draw the circuit described by the following Verilog module. Completely label all signals and gates, including propagation delays. The module can be replace with another primitive gate in Verilog. What is that primitive gate?

```
module exercise10
    (output       f,
    input         x, y);
    nand #10
            g1 (f1, x, x),
            g2 (f2, y, y),
            g3 (f3, x, y),
            g4 (f4, f1, f2),
            g5 (f, f3, f4);
endmodule
```

14.11 Rewrite Figure 14.22 using a one-hot state encoding.

14.12 Write a Verilog description using (i) a single always block and (ii) using two always blocks, one for the combinational logic and the other for the sequential, for the FSM shown in Figure 14.35.

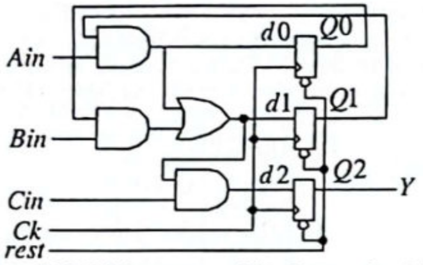


Figure 12.35 Finite state machine for exercise 14.12.

14.13 Design, implement, and simulate a 4-bit 2's complement adder/subtractor.

14.14 Write a procedural Verilog implementation of the finite state machine of Figure 12.36. Simulate and test your circuit for an input sequence (read left to right) of 100101101111. Display the output sequence of your fsm in decimal and relate it back to your input sequence.

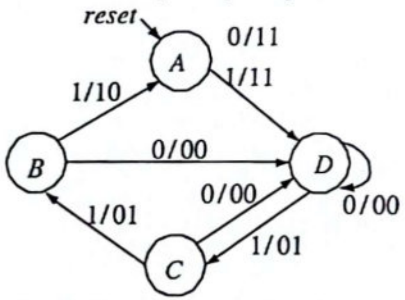


Figure 12.36 Finite state machine for exercise 14.14.

14.15 A sequential circuit has one input, X, and one output, Y. The output is 1 every time the input sequence has exactly two or exactly four 1s in a row. Otherwise, the output is 0. Example inputs and corresponding outputs are shown below:

Input: 0110111011110111110
Output: 0001000000001000000

Write a Verilog implementation for the machine. Simulate and test the circuit using the input sequence given above.

Further Readings

1. J. Bhasker, *Verilog HDL Synthesis*, Star Galaxy Press, 1998.
2. M. D. Ciletti, Modeling, Synthesis, and Rapid Prototyping with Verilog HDL, Prentice Hall, 1999.
3. M. Morris Mano, *Digital Design*, 3rd ed., Pearson Education, 2002.
4. Donald E. Thomas and Philip R. Moorby, *The Verilog Hardware Description Language*, 5th ed., Kluwer Academic Publishers, 2003.

CHAPTER 15
LABORATORY EXPERIMENTS

15.0 Introduction

Laboratory experiments in digital logic design are presented in this chapter. The experiments provide hands-on experience for the students using this book. The experiments are ordered according to the material presented in this book.

A digital logic trainer suitable for performing the experiments must have the following equipment:

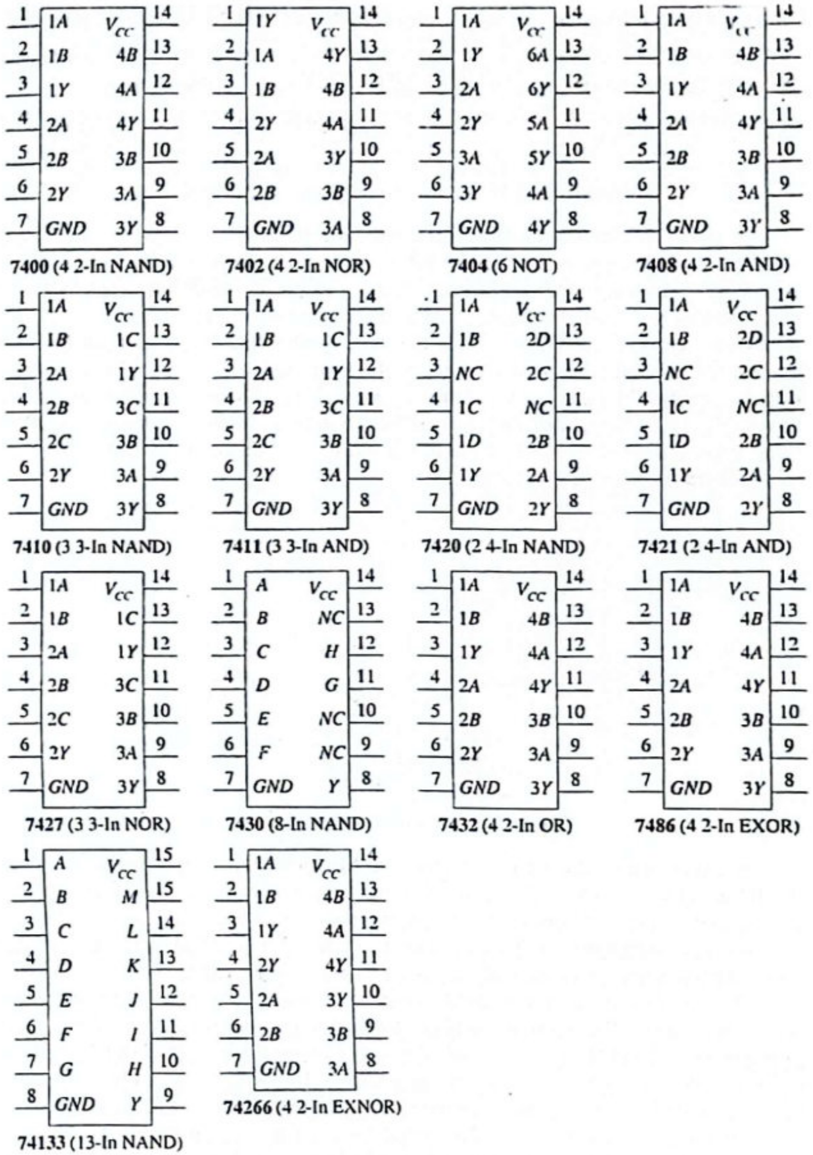
1. LED (light-emitting diode) indicator lamps
2. 7-segment numeric displays
3. Data switches to provide logic 0 and 1 signals
4. Pulsers with push-button switches and debounce circuits to generate single pulses
5. A clock-pulse generator with variable frequency ranges
6. A power supply of 5V for TTL ICs
7. Socket strips for mounting ICs
8. Solid hookup wire and a pair of wire strippers for cutting the wires

Additional equipment required are

1. A logic probe to be used for debugging
2. A number of TTL ICs (series 7400)

The integrated circuits (ICs) to be used in the experiments can be classified as small-scale-integration (SSI) or medium-scale integration (MSI) circuits. SSI circuits contain individual gates or flip-flops, and MSI circuits perform specific digital functions. The SSI gate ICs needed for the experiments are shown in Figure 15.1 and the SSI flip-flop ICs needed are shown in Figure 15.2. The pin assignment for the ICs is indicated in the diagrams. The pin marked V_{CC} and *GND* are the supply terminals, which must be connected to a power supply of 5V for proper operation. In some ICs, some of the pins are not used by the circuits and are marked as *NC* (no connection). Each IC is recognized by its identification number; for example, the 2-input NAND gates are found inside the IC whose number is 7400.

The MSI ICs needed for the experiments are introduced and explained when they are used for the first time. The information given in this chapter about the MSI circuits should be sufficient for performing the experiments adequately. Nevertheless, a reference to a *Data Book* will always be preferable, as it gives more detailed description of the circuits.


1 1A Vcc 14
2 1B 4B 13
3 1Y 4A 12
4 2A 4Y 11
5 2B 3B 10
6 2Y 3A 9
7 GND 3Y 8
1 1Y Vcc 14
2 1A 4Y 13
3 1B 4B 12
4 2Y 4A 11
5 2A 3Y 10
6 2B 3B 9
7 GND 3A 8
1 1A Vcc 14
2 1Y 6A 13
3 2A 6Y 12
4 2Y 5A 11
5 3A 5Y 10
6 3Y 4A 9
7 GND 4Y 8
1 1A Vcc 14
2 1B 4B 13
3 1Y 4A 12
4 2A 4Y 11
5 2B 3B 10
6 2Y 3A 9
7 GND 3Y 8
1 1A Vcc 14
2 1B 1C 13
3 2A 1Y 12
4 2B 3C 11
5 2C 3B 10
6 2Y 3A 9
7 GND 3Y 8
1 1A Vcc 14
2 1B 1C 13
3 2A 1Y 12
4 2B 3C 11
5 2C 3B 10
6 2Y 3A 9
7 GND 3Y 8
1 1A Vcc 14
2 1B 2D 13
3 NC 2C 12
4 1C NC 11
5 1D 2B 10
6 1Y 2A 9
7 GND 2Y 8
1 1A Vcc 14
2 1B 2D 13
3 NC 2C 12
4 1C NC 11
5 1D 2B 10
6 1Y 2A 9
7 GND 2Y 8
1 1A Vcc 14
2 1B 1C 13
3 2A 1Y 12
4 2B 3C 11
5 2C 3B 10
6 2Y 3A 9
7 GND 3Y 8
1 A Vcc 14
2 B NC 13
3 C H 12
4 D G 11
5 E NC 10
6 F NC 9
7 GND Y 8
1 1A Vcc 14
2 1B 4B 13
3 1Y 4A 12
4 2A 4Y 11
5 2B 3B 10
6 2Y 3A 9
7 GND 3Y 8
1 1A Vcc 14
2 1B 4B 13
3 1Y 4A 12
4 2A 4Y 11
5 2B 3B 10
6 2Y 3A 9
7 GND 3Y 8
1 A Vcc 15
2 B M 15
3 C L 14
4 D K 13
5 E J 12
6 F I 11
7 G H 10
8 GND Y 9
1 1A Vcc 14
2 1B 4B 13
3 1Y 4A 12
4 2Y 4Y 11
5 2A 3Y 10
6 2B 3B 9
7 GND 3A 8

7400 (4 2-In NAND) **7402** (4 2-In NOR) **7404** (6 NOT) **7408** (4 2-In AND)

7410 (3 3-In NAND) **7411** (3 3-In AND) **7420** (2 4-In NAND) **7421** (2 4-In AND)

7427 (3 3-In NOR) **7430** (8-In NAND) **7432** (4 2-In OR) **7486** (4 2-In EXOR)

74133 (13-In NAND) **74266** (4 2-In EXNOR)

Figure 15.1 Pin Connection of some logic gate ICs.

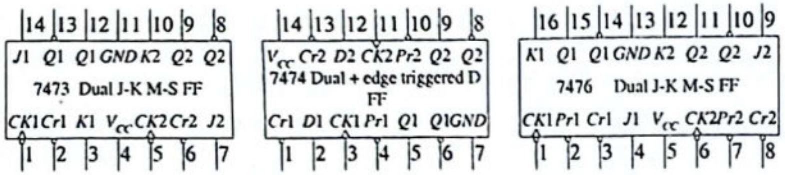


Figure 15.2 Pin connection of some flip-flop ICs.

In building circuits for the experiments, the IC is inserted into a socket, and wires are connected to the various pins through the socket terminals.

When drawing logic diagrams with SSI gates and flip-flops for the experiments, we draw the diagram using logic symbols of gates and flip-flops, the IC number is written inside or beside the logic symbol and the chosen pin numbers are written along the input/output lines of the logic symbols. As a SSI IC contains a limited number of gates or flip-flops, in many logic diagrams more than one IC having the same IC number may be needed. To distinguish the individual ICs of the same type, the serial number of the ICs is written beside the IC number. A sample logic diagram is shown in Figure 15.3.

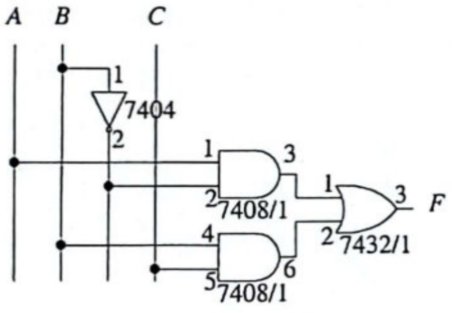


Figure 15.3 A sample logic diagram.

When drawing schematic diagrams with MSI ICs for the experiments, we show the IC in a block diagram form, the IC number is written inside the block, and the corresponding pin numbers are written along the external lines.

It is characteristic of TTL circuits that an input terminal with no external connection has the effect of producing a signal equivalent to logic 1.

The rest of this chapter is divided into 16 sections, with each section covering one experiment. The section number designates the experiment number. Each experiment is divided into two parts – pre-lab work and lab work. Pre-lab work of the experiment must be done prior to the lab class. Lab work of the experiment is to be done in the laboratory, which should take about two hours of laboratory time.

All circuit connections on the digital logic trainer should be made with the power supply switches of the digital logic trainer in the *off* position. After making sure that all the connections are correct, turn the power switches in the *on* position. Any alteration of the circuit should also be made with the power supply switches in the off position.

15.1 Binary and BCD Numbers

Objectives

1. To have an introduction about the digital logic trainer used in the laboratory.
2. To practice generating binary number sequence with the data switches and to display them using LED indicators.
3. To practice generating BCD digits with the data switches and to display them using 7-segment display.

Theory

Binary and BCD numbers are introduced in Sections 2.4 and 2.8, respectively.

Pre-Lab Work

Generating 4-bit Binary Number Sequence

1. Prepare a list of 4-bit binary numbers in a table similar to that shown in Table 15.1.

Table 15.1 4-bit binary numbers and their decimal equivalents.

Binary Number $B_3B_2B_1B_0$	Equivalent Decimal Number

Generating BCD Digits

1. Prepare a list of BCD codes in a table similar to that shown in Table 15.2.

Table 15.2 BCD codes

Decimal Digit	BCD Code $B_3B_2B_1B_0$

Lab Work

Introduction about Digital Logic Trainer

1. With the help of your instructor and lab assistants, carefully observe the functionality and accessories of the digital logic trainer used in your lab.

Generating 4-bit Binary Number Sequence

1. Designate four data switches as $B_3B_2B_1B_0$ (the LSB B_0 switch should be at your right side) and also designate four LED indicators as $B_3B_2B_1B_0$ (the LSB B_0 indicator should be at your right side). Connect the data switches to their corresponding LED indicators.
2. Generate the binary number sequence of Table 15.1 using the data switches sequentially and observe the LED indicators.

Generating BCD Digits

1. Designate four data switches as $B_3B_2B_1B_0$. Connect the data switches to the input of the BCD-to-7-segment decoder/driver inputs in the proper order of their LSB and MSB.
2. Generate the BCD code sequence of Table 15.2 using data switches sequentially and observe the 7-segment display.

15.2 Digital Logic Gates

Objectives

1. To obtain and verify the truth tables of NOT gate and 2-input AND, OR, NAND, NOR, and EXOR gate ICs.
2. To implement NOT gate and 2-input AND, OR, and NOR gates using 2-input NAND gate.
3. To implement NOT gate and 2-input AND, OR, and NAND gates using 2-input NOR gate.

Theory

NOT gate and 2-input AND, OR, NAND, NOR, and EXOR gates are introduced in Section 4.3. The logic symbols and truth tables of these digital logic gates are given in Figure 4.1. The pin assignments of these gates are shown in Figure 15.1.

Universality of NAND and NOR gates is introduced in Section 4.4. Implementation of NOT gate and 2-input AND and OR gates using 2-input NAND and NOR gates is shown in Figure 4.6 and 4.7, respectively.

Implementation of a 2-input NOR gate using 2-input NAND gates and implementation of a 2-input NAND gate using 2-input NOR gates can be performed with the help of DeMorgan's theorem (see Section 3.3) and the concept of implementing NOT, AND, and OR gates using NAND and NOR gates. For example, $(AB)' = A' + B'$. Therefore, a 2-input NAND gate can be implemented using 2-input NOR gate by first complementing A and B by two NOR gates and then ORing them by an OR gate implemented with NOR gates. In a similar manner, a 2-input NOR gate can be implemented using 2-input NAND gates.

Pre-Lab Work

Obtaining and verifying truth tables of digital logic gates

1. Prepare truth table of NOT gate and 2-input AND, OR, NAND, NOR, and EXOR gates.

Implementing NOT gate and 2-input AND, OR, and NOR gates using 2-input NAND gate

1. Draw the logic diagrams for implementing NOT gate and 2-input AND, OR, and NOR gates using 2-input NAND gates.

Implementing NOT gate and 2-input AND, OR, and NAND gates using 2-input NOR gate

1. Draw the logic diagram for implementing NOT gate and 2-input AND, OR, and NAND gates using 2-input NOR gates.

Lab Work

ICs required

7400 Quadruple 2-input NAND gates
7402 Quadruple 2-input NOR gates
7404 Hex Inverters (NOT gates)
7408 Quadruple 2-input AND gates
7432 Quadruple 2-input OR gates
7486 Quadruple 2-input EXOR gates

Obtaining and verifying truth tables of digital logic gates

1. Take an IC from the above list. Mount the IC on the breadboard of the digital logic trainer. Connect +5V to V_{CC} pin and ground to *GND* pin of the IC.
2. Use one gate from the IC. Connect the input pin(s) to data switch(es) and the output pin to an LED indicator on the digital logic trainer.
3. With the data switches vary the input(s) to the gate and observe the output. In this way prepare the truth table of the gate and verify it.
4. Repeat the above steps for all ICs listed above.

Implementing NOT gate and 2-input AND, OR, and NOR gates using 2-input NAND gate

1. Construct the circuit for implementing NOT gate using 2-input NAND gate and verify its truth table.
2. Repeat step 1 for implementation of 2-input AND, OR, and NOR gates using 2-input NAND gate.

Implementing NOT gate and 2-input AND, OR, and NAND gates using 2-input NOR gate

1. Construct the circuit for implementing NOT gate using 2-input NOR gate and verify its truth table.
2. Repeat step 1 for implementation of 2-input AND, OR, and NAND gates using 2-input NOR gate.

15.3 EXOR Gate Implementation

Objectives

1. To implement 2-input EXOR gate using NOT gate and 2-input AND and OR gates.
2. To implement 2-input EXOR gate using 2-input NAND gates.
3. To implement 2-input EXOR gate using 2-input NOR gates.

Theory

Various implementation of 2-input EXOR gate is shown in Figure 4.9.

Pre-Lab Work

1. Draw circuits for implementing 2-input EXOR gate using (a) NOT gate and 2-input AND and OR gates, (b) 2-input NAND gates, and (c) 2-input NOR gates.

Lab Work

ICs required

7400 Quadruple 2-input NAND gates
7402 Quadruple 2-input NOR gates
7404 Hex Inverters (NOT gates)
7408 Quadruple 2-input AND gates
7432 Quadruple 2-input OR gates

Implementing EXOR gate

1. Construct the circuit for implementing a 2-input EXOR gate using NOT gates and 2-input AND and OR gates on the breadboard of the digital logic trainer. Connect the two inputs of the circuit to two data switches and the output to an LED indicator. Obtain and verify the truth table of the constructed circuit.
2. Repeat step 1 for implementing 2-input EXOR gate using 2-input NAND gates and 2-input NOR gates, respectively.

15.4 Simplification and Implementation of Boolean Function

Objectives

1. To obtain simplified sum of products expression of a Boolean function using K-map.
2. To draw logic diagram of a Boolean function.
3. To implement a Boolean function using logic gates.

Theory

Logic diagram of a Boolean function is introduced in Section 3.4. Canonical sum of products and noncanonical sum of products expression of a Boolean function are introduced in Section 3.5. Simplification of Boolean functions using K-map is introduced in Section 5.4.

Pre-Lab Work

1. The canonical sum of products expression of an arbitrary Boolean function is given as $F(A,B,C,D)=\sum(4,5,12,13,15)$. Obtain its canonical sum of products expression.
2. Simplify the given function using K-map. Obtain the simplified sum of products expression.
3. Draw logic diagrams for both the canonical and simplified sum of products expressions of the given function.

Lab Work

ICs required

7404 Hex Inverters (NOT gates)
7408 Quadruple 2-input AND gates
7411 Triple 3-input AND gates
7421 Dual 4-input AND gates
7432 Quadruple 2-input OR gates

Implementing Boolean function using logic gates

1. Construct both the logic diagrams using logic gates on a digital logic trainer. Make the inputs of the two circuits common so that operation of the two circuits can be observed simultaneously for the same input value. Connect the four inputs of the circuits to four data switches and the two outputs of the two circuits to two LED indicators.
2. Obtain truth tables for the two circuits and verify that the two truth tables are same. Also verify that minterms 4, 5, 12, 13, and 15 are 1 in the truth tables and the remaining minterms are 0.

15.5 Design and Implementation of a Combinational Circuit

Objectives

1. To design a combinational circuit from descriptive problem specification.
2. To implement a combinational circuit using AND-OR, OR-AND, NAND-NAND, and NOR-NOR logic.

Theory

The design procedure of a combinational circuit is discussed in Section 6.2. Implementation of a combinational circuit using NAND-NAND and NOR-NOR circuits are discussed in Sections 6.4 and 6.5, respectively.

Pre-Lab Work

Problem specification: In a combinational logic circuit, four input lines *A*, *B*, *C*, and *D* are being used to represent a 4-bit binary number with *A* as the MSB and *D* as the LSB. The circuit produces a high output only when the binary number is greater than 0101_2. Design and implement the circuit.

1. Prepare a truth table for the given combinational circuit. Represent the truth table in a 4-variable K-map.
2. Write simplified sum of products function from the K-map. Draw the AND-OR logic diagram for the simplified sum of products function.
3. Draw NAND-NAND logic diagram for the simplified sum of products function.
4. Write simplified product of sums function from the K-map. Draw OR-AND logic diagram for the simplified product of sums function.
5. Draw NOR-NOR logic diagram for the simplified product of sums function.

Lab Work

ICs required

7400 Quadruple 2-input NAND gates
7402 Quadruple 2-input NOR gates
7408 Quadruple 2-input AND gates
7432 Quadruple 2-input OR gates

Implementing Combinational Circuit

1. Construct all four logic diagrams on a digital logic trainer. Make inputs of all the circuits common so that outputs of all circuits can be observed for the same input value. Connect four inputs to four data switches and four outputs of the four circuits to four LED indicators.
2. Construct truth tables for the four circuits and verify them. Observe that truth tables of all the four circuits are same.

15.6 Binary Adders and Subtractors

Objectives

1. To implement and test a full adder using random gates.
2. To test the operation of MSI IC type 7483 (4-bit binary parallel adder).
3. To implement and test a 4-bit 2's complement adder/subtractor using MSI IC type 7483 (4-bit binary parallel adder).

Theory

Design of full-adder using random gates is discussed in Section 7.2. The MSI IC type 7483 (4-bit binary parallel adder) is introduced in Section 7.8. Design of a 4-bit 2's complement adder/subtractor using a 4-bit binary parallel adder is discussed in Section 7.6.

Pre-Lab Work

Implementing and testing a full adder using random gates

1. Draw the logic diagram of a full-adder using EXOR gates.
2. Write the truth table of a full-adder.

Testing the operation of MSI IC type 7483 (4-bit binary parallel adder)

1. Learn the pin assignment of IC type 7483 (4-bit binary parallel adder) as shown in Figure 15.4. The two 4-bit input binary numbers are A_3 through A_B and B_0 through B_3. The 4-bit sum is obtained from S_0 through S_3. C_0 is the input carry and C_4 the output carry.

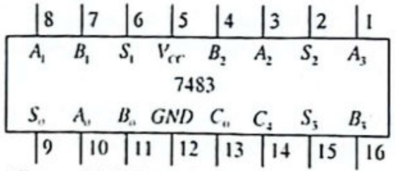


Figure 15.4 Pin assignment of IC type 7483.

Implementing and testing a 4-bit 2's complement adder/subtractor using an MSI IC type 7483 (4-bit binary parallel adder)

1. Draw the schematic diagram for implementing a 4-bit 2's complement adder/subtractor using IC type 7483 (4-bit binary parallel adder). Write the operation of the circuit explaining how the mode selector input M determines the mode of operations as adder or subtractor.

Lab Work

ICs required

7408 Quadruple 2-input AND gates
7432 Quadruple 2-input OR gates
7486 Quadruple 2-input EXOR gates
7483 4-bit binary parallel adder

Implementing and testing a full adder using random gates

1. Construct the logic diagram for a full-adder using SSI ICs on the breadboard. Connect the three inputs of the circuit to three data switches and two outputs to two LED indicators.
2. Prepare the truth table of the full-adder circuit and verify its operation.

Testing the operation of MSI IC type 7483 (4-bit binary parallel adder)

1. Connect the four A inputs of IC type 7483 to a fixed binary number 1001 (decimal 9) and the four B inputs and the input carry C_0 to five data switches. Connect the four S outputs and the output carry C_4 to five LED indicators.
2. Perform addition of some binary numbers and input carry with the fixed binary number 1001 and check that the output sum and output carry give the proper value.

Implementing and testing a 4-bit 2's complement adder/subtractor using an MSI IC type 7483 (4-bit binary parallel adder)

1. Construct the schematic diagram for a 4-bit 2's complement adder/subtractor using IC type 7483 and other SSI gates. Connect the four A inputs to a fixed binary number 001 (decimal 3) and the four B inputs to data switches. Connect the mode select input M to a data switch. Connect the four sum outputs S and carry output C_4 to five LED indicators.
2. Perform the following addition/subtraction operations and record the values of the outputs S_0 through S_3 and the output carry C_4. Verify the results.

$3+4$	$3-4$
$3+3$	$3-3$
$3+2$	$3-2$

15.7 Decoder and Its Use in Combinational Logic Implementation

Objectives

1. To implement and test a 2-to-4-lines decoder with active-LOW outputs and active-LOW enable input using random gates.
2. To test the operation of an MSI IC type 74138 (3-to-8-lines decoder with active-LOW outputs).
3. To implement and test combinational logic functions using an MSI IC type 74138 (3-to-8-lines decoder with active-LOW outputs) and external NAND gates.

Theory

Design of decoders using random gates is discussed in Section 8.3. The MSI IC type 74138 (3-to-8-line decoder with active-LOW outputs) is introduced in Section 8.3. Combinational logic implementation using decoders is also discussed in Section 8.3.

Pre-Lab Work

Implementing and testing a 2-to-4-lines decoder with active-LOW outputs and active-LOW enable input using random gates

1. Draw the logic diagram for a 2-to-4-lines decoder with active-LOW outputs and active-LOW enable input.
2. Write the truth table of the above decoder.

Testing the operation of MSI IC type 74138 (3-to-8-lines decoder with active-LOW outputs)

1. Learn the pin assignment of IC type 74138 (3-to-8-lines decoder with active-LOW outputs) as shown in Figure 15.5. A_o through A_2 are three address lines and O_o through O_7 are eight active-LOW outputs. It has two active-LOW (E_1 and E_2) and one active-HIGH (E_3) enable inputs internally ANDed to produce a single enable signal.

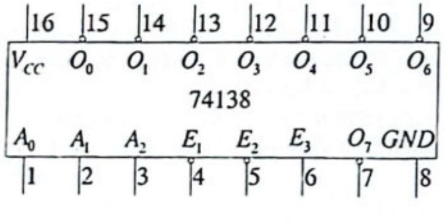


Figure 15.5 Pin assignment of IC type 74138

2. Draw the truth table of the MSI IC type 74138.

Implementing and testing combinational logic functions using the MSI IC type 74138 (3-to-8-lines decoder with active-LOW outputs) and external NAND gates

1. Draw the schematic diagram for implementing the following two combinational logic functions using the MSI IC type 74138 (3-to-8-lines decoder with active-LOW outputs) and external NAND gates.

$$F_1(A,B,C) = \Sigma(1,2,4,7)$$

$$F_2(A,B,C) = \Sigma(0,5,6,7)$$

Lab Work

ICs required

7404 Hex Inverters (NOT gates)
7410 Triple 3-input NAND gates
7420 Dual 4-input NAND gates
74138 3-to-8-line decoder with active-LOW outputs

Implementing and testing a 2-to-4-lines decoder with active-LOW outputs and active-LOW enable input using random gates

1. Construct the logic diagram for a 2-to-4-lines decoder with active-LOW outputs and active-LOW enable input using random gates. Connect the three inputs (A_1, A_0, and E) of the circuit to three data switches and four outputs to four LED indicators.
2. Apply binary 00 through 11 to A_1 A_0 address inputs and observe the outputs by changing E enable input. Verify the operation.

Testing the operation of MSI IC type 74138 (3-to-8-lines decoder with active-LOW outputs)

1. Connect enable inputs E_1 and E_2 to fixed 0s and E_3 to a data switch. Use E_3 as active-HIGH enable input. Connect address lines A_0 through A_2 to three data switches. Connect outputs O_0 through O_7 to eight LED indicators.
2. Apply binary number 000 through 111 to address lines $A_2A_1A_0$ and observe, by changing the enable input E_3, the outputs. Check that, when the IC is enabled, the selected output is 0 and the remaining outputs are 1.

Implementing and testing combinational logic functions using the MSI IC type 74138 (3-to-8-lines decoder with active-LOW outputs) and external NAND gates

1. Construct the circuits for implementing the given combinational logic functions. Connect enable inputs E_1 and E_2 to fixed 0s and E_3 to a data switch. Use E_3 as active-HIGH enable input. Connect the input lines ABC ($A_2A_1A^0$) to three data switches. Connect the two function outputs to two LED indicators.
2. Prepare the truth tables of the circuit and verify that it implements the two given combinational logic functions.

15.8 Code Converters

Objectives

1. To implement and test BCD-to-Excess-3 code converter using MSI IC type 7483 (4-bit binary parallel adder).
2. To test the operation of MSI IC type 7447 (7-segment display decoder/driver with active-LOW, open collector outputs).

Theory

BCD-to-Excess-3 code converter is introduced in Section 8.4. The MSI IC type 7483 (4-bit binary parallel adder) is introduced in Section 7.8. The pin assignment of the IC 7483 is also given in Figure 15.4. BCD-to-7-segment display decoder is introduced in Section 8.5. The MSI IC type 7447 (7-segment display decoder/driver with active-LOW, open collector outputs) is also introduced in Section 8.5.

The common-anode 7-segment display IC is used for displaying any one of the decimal digits 0 through 9 with the 7-segment display decoder/driver with active-LOW outputs. It contains seven LED (light-emitting diode) segments on top of the package. Anodes of all the LEDs are common.

Pre-Lab Work

Implementing and testing BCD-to-Excess-3 code converter using MSI IC type 7483 (4-bit binary parallel adder)

1. Write the truth table of BCD-to-Excess-3 code converter.
2. Draw the logic diagram of BCD-to-Excess-3 code converter using MSI IC type 7483 (4-bit binary parallel adder).

Testing the operation of MSI IC type 7447 (7-segment display decoder/driver with active-LOW, open collector outputs)

1. Learn the pin assignment of IC type 7447 (7-segment display decoder/driver with active-LOW, open collector outputs) as shown in Figure 15.6. The four BCD inputs are $B_0 - B_3$ and the seven active-LOW, open collector outputs are $a' - g'$. If the input data is decimal zero, $RBI' = 0$ blanks the display. $BI/RBO' = 0$ unconditionally turns off all segment outputs. $LT' = 0$ turns on all segment outputs, provided that $BI/RBO' = 1$.

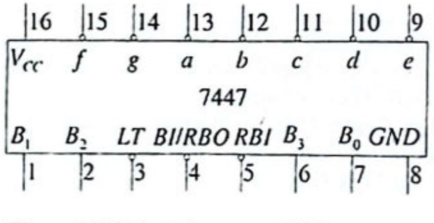


Figure 15.6 Pin assignment of IC type 7447.

2. Learn the connection of the IC type 7447 with the common-anode 7-segment display as shown in Figure 15.7.

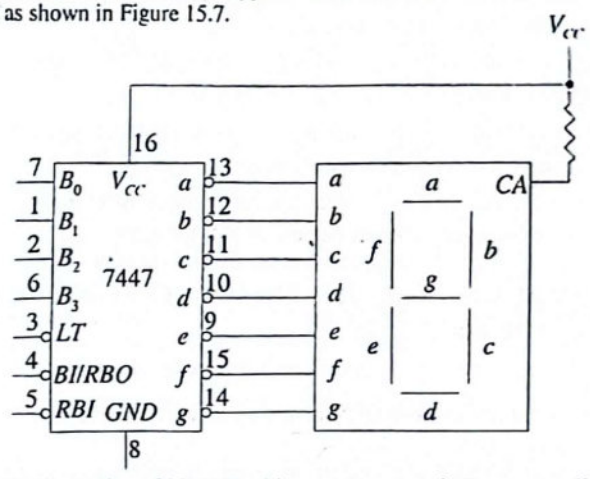


Figure 15.7 Connection of IC 7447 with a common-anode 7-segment display.

Lab Work

ICs required

1. IC 7483 4-bit binary parallel adder
2. IC 7447 BCD-to-7-segment display decoder/driver with active-LOW, open-collector output
3. Common-Anode 7-segment display

Implementing and testing BCD-to-Excess-3 code converter using MSI IC type 7483 (4-bit binary parallel adder)

1. Construct the logic diagram of BCD-to-Excess-3 code converter using MSI IC type 7483 (4-bit binary parallel adder) on the breadboard. Connect the four inputs to four data switches and the four outputs to four LED indicators.
2. Apply BCD inputs 0000 through 1001 and observe the Excess-3 outputs. Verify the truth table of the circuit.

Testing the operation of MSI IC type 7447 (7-segment display decoder/driver with active-LOW, open collector outputs)

1. Connect the IC 7447 (BCD-to-7-segment display decoder/driver with active-LOW, open collector outputs) and the common-anode 7-segment display as in Figure 15.7. Connect the four BCD inputs $B_0 - B_3$ to four data switches. Also connect LT', BI/RBO', and RBI' inputs to three more data switches.
2. Set $LT' = 1$, $BI/RBO' = 1$, and $RBI' = 1$. Apply BCD inputs 0000 through 1001 and observe the decimal display from 0 to 9.

3. Apply invalid BCD inputs 1010 through 1111 and observe the output display patterns. These patterns are meaningless.
4. Set $LT' = 1$, $BI/RBO' = 1$, and $RBI' = 0$. Apply 0000 input. The display will be blank. For other BCD inputs, output will be normal.
5. Set $LT' = 1$, $RBI' = 1$, and $BI/RBO' = 0$. Apply BCD inputs 0000 through 1001. Observe that all outputs are blank for all inputs.
6. Set $LT' = 0$, $BI/RBO' = 1$, and $RBI' = 1$. Apply BCD inputs0000 through 1001. Observe that all segment outputs are ON for all inputs.

15.9 Multiplexer and Its Use in Combinational Logic Implementation

Objectives

1. To implement and test a 4-to-1-line multiplexer with active-LOW enable input using random gates.
2. To test the operation of MSI IC type 74151 (8-to-1-line multiplexer with active-LOW enable input).
3. To implement and test combinational logic function using MSI IC type 74151 (8-to-1-line multiplexer with active-LOW enable input).

Theory

Design of multiplexers using random gates is discussed in Section 8.7. The IC type 74151 (8-to-1-line multiplexer with active-LOW enable input) is introduced in Section 8.7. Combinational logic implementation using multiplexers is also discussed in Section 8.7.

Pre-Lab Work

Implementing and testing a 4-to-1-line multiplexer with active-LOW enable input using random gates

1. Write the truth table of a 4-to-1-line multiplexer with active-LOW enable input.
2. Draw the logic diagram of a 4-to-1-line multiplexer with active-LOW enable input using random gates.

Testing the operation of IC type 74151 (8-to-1-line multiplexer with active-LOW enable input)

1. Learn the pin assignment of IC type 74151 (8-to-1-line multiplexer with active-LOW enable input) as shown in Figure 15.8. The eight data inputs are designated I_0 through I_7. The three address or selection inputs A_0 through A_2 select the particular data input to be multiplexed and applied to the output. The IC provides both uncomplemented output O and complemented output O'. The IC also has an active-LOW enable input E.

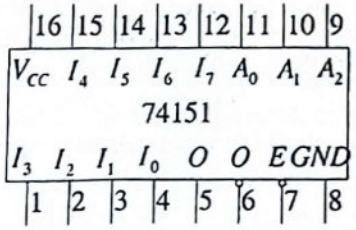

16 15 14 13 12 11 10 9
V_{CC} I_4 I_5 I_6 I_7 A_0 A_1 A_2
74151
I_3 I_2 I_1 I_0 O O E GND
1 2 3 4 5 6 7 8

Figure 15.8: Pin assignment of IC 74151.

Implementing and testing combinational logic function using IC type 74151 (8-to-1-line multiplexer with active-LOW enable input)

1. Prepare the implementation table for implementing the combinational logic function $F(A,B,C,D) = \Sigma(0,3,4,6,8,11,13,15)$ using an 8-to-1-line multiplexer.
2. Draw the logic diagram for implementing the logic function using IC type 74151 (8-to-1-line multiplexer with active-LOW enable input).

Lab Work

ICs required

7404 Hex Inverters (NOT gates)
7421 Double 4-input AND gates
7432 Quadruple 2-input OR gates
74151 8-to-1-line multiplexer with active-LOW enable input

Implementing and testing a 4-to-1-line multiplexer with active-LOW enable input using random gates

1. Construct the logic diagram for a 4-to-1-line multiplexer with active-LOW enable input using random gates. Connect the four data inputs I_0 through I_0 to an arbitrary 4-bit binary number. Connect A_1, A_0, and E to three switches and the output O to an LED indicator.
2. Apply binary 00 through 11 to address lines A_1 A_0 and observe the outputs by changing E input. Check that, when the circuit is enabled, the output is equal to the selected data input.

Testing the operation of IC type 74151 (8-to-1-line multiplexer with active-LOW enable input)

1. Connect the eight data inputs I_0 through I_7 to an arbitrary 8-bit binary number. Connect the address lines A_2 through A_0 to three data switches. Connect the active-LOW enable input E to a data switch. Connect the uncomplemented output O to an LED indicator.

2. Apply binary 000 through 111 to address lines $A_2A_1A_0$ and observe outputs by changing the enable input E. Check that, when the IC is enabled, the output is equal to the selected data input.

Implementing and testing combinational logic function using IC type 74151 (8-to-1-line multiplexer with active-LOW enable input)

1. Construct the logic diagram for implementing the logic function $F(A,B,C,D) = \Sigma(0,3,4,6,8,11,13,15)$ using IC type 74151 (8-to-1-line multiplexer with active-LOW enable input).
2. Prepare the truth table of the circuit and verify that it implements the given combinational logic function.

15.10 Flip-Flops

Objectives

1. To implement and test a J-K flip-flop using random gates.
2. To test the operation of SSI IC type 7476 (dual J-K master-slave flip-flop).
3. To implement and test D and T flip-flops Using SSI IC type 7476 (dual J-K master-slave J-K flip-flop).

Theory

Design of J-K flip-flop using random gates is discussed in Section 9.4. The SSI IC type 7476 (dual J-K master-slave flip-flops) is introduced in Section 9.9. Implementation of D and T flip-flops using J-K flip-flops are discussed in Sections 9.5 and 9.6, respectively.

Pre-Lab Work

Implementing and testing a J-K flip-flop using random gates

1. Draw the logic diagram for implementing a J-K flip-flop using NAND gates.
2. Write the truth table of a J-K flip-flop.

Testing the operation of IC type 7476 (dual J-K master-slave flip-flop)

1. Learn the pin assignment of IC type 7476 (dual J-K master-slave flip-flops) as shown in Figure 15.9. The IC has two separate J-K master-slave flip-flops.

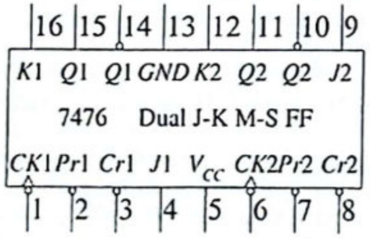


Figure 15.9 Pin assignment of IC 7476.

Implementing and testing D and T flip-flops Using IC type 7476 (dual J-K master-slave flip-flops)

1. Draw the logic diagram for implementing D flip-flop using IC type 7476 (dual J-K master-slave flip-flops).
2. Write the truth table of D flip-flop.
3. Draw the logic diagram for implementing T flip-flop using IC type 7476 (dual J-K master-slave flip-flops).
4. Write the truth table of T flip-flop.

Lab Works

ICs required

7410 Triple 3-input NAND gates
7404 Hex NOT gates
7476 Dual J-K master-slave flip-flops

Implementing and testing a J-K flip-flop using random gates

1. Construct the logic diagram for J-K flip-flop using NAND gates. Connect J, K, Pr, and Cr inputs to four data switches. Connect CK input to a push-button pulser. Connect Q and Q' outputs to two LED indicators.
2. Asynchronously set the value of Q to 0 by setting $Pr = 1$, $Cr = 0$, and $CK = 0$. Also asynchronously set the value of Q to 1 by setting, $Pr = 0$, $Cr = 1$, and $CK = 0$.
3. Set $Pr = Cr = 1$ for normal operation. Verify the truth table of J-K flip-flop.

Testing the operation of IC type 7476 (dual J-K master-slave flip-flop)

1. Choose any flip-flop from the IC type 7476. Connect J, K, Pr, and Cr inputs to four data switches. Connect CK input to a push-button pulser. Connect Q and Q' outputs to two LED indicators.
2. Asynchronously set and reset the flip-flop.
3. Verify the truth table of the flip-flop.

Implementing and testing D and T flip-flops Using IC type 7476 (dual J-K master-slave flip-flops)

1. Construct the logic diagram for D flip-flop using IC type 7476 (dual J-K master-slave flip-flops).
2. Verify the truth table of D flip-flop from the constructed circuit.
3. Construct the logic diagram for T flip-flop using IC type 7476 (dual J-K master-slave flip-flops).
4. Verify the truth table of T flip-flop from the constructed circuit.

15.11 Design and Implementation of a Synchronous Sequential Circuit.

Objectives

1. To design and test a synchronous sequential circuit from the problem specification.

Theory

The design procedure of synchronous sequential circuit is discussed in Section 10.5. Other related topics are discussed in Sections 10.2 through 10.4.

Pre-Lab Work

Problem specification: The transition diagram of a synchronous sequential circuit is given in Figure 15.10. The circuit requires two flip-flops and has one input line and one output line.

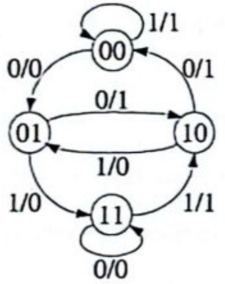


Figure 15.10 Transition diagram for Experiment 15.11.

1. Designate the two flip-flops as *FF*1 and *FF*0. Designate the input as x and the output as y. Prepare the transition table.
2. Using J-K flip-flops, prepare the flip-flop excitation table and output table.
3. Using K-map, derive the flip-flop excitation equations and output equation.
4. Draw the logic diagram.

Lab Work

1. Construct the logic diagram. Connect the input x to a data switch and the output y to an LED indicator. Connect the *CK* input of the flip-flops to a push-button pulser. Connect the *Pr* and *Cr* inputs of the flip-flops to two data switches. Connect the outputs of the two flip-flops Q_1Q_0 to two LED indicators.
2. Asynchronously clear the two flip-flops.
3. Prepare the transition diagram of the implemented circuit and verify it with Figure 15.10.

15.12 Registers

Objectives

1. To implement and test a 4-bit serial-in serial-out right-shift register using D flip-flops.
2. To test the operation of MSI IC type 74194 (4-bit bi-directional universal register).
3. To implement and test a 4-bit ring counter using MSI IC type 74194 (4-bit bi-directional universal register).

Theory

Design of serial-in serial-out registers using flip-flops is discussed in Section 11.3. Universal register is introduced in Section 11.7. The IC type 74194 (4-bit bi-directional universal register) is introduced in Section 11.8. Ring counter is discussed in Section 11.14.

Pre-Lab Work

Implementing and testing a 4-bt serial-in serial-out right-shift register using D flip-flops

1. Draw the logic diagram of a 4-bit serial-in serial-out right-shift register using IC type 7474 (dual +edge triggered D flip-flops).

Testing the operation of IC type 74194 (4-bit bi-directional universal register)

1. Learn the pin assignment of IC type 74194 (4-bit bi-directional universal register) as shown in Figure 15.11. The parallel data inputs are $D_0D_1D_2D_3$ and the parallel outputs are $D_0D_1D_2D_3$. The serial inputs are D_{SL} for left-shift and D_{SR} for right-shift. The master reset input $MR' = 0$ clears all the outputs $Q_0Q_1Q_2Q_3$. $MR' = 1$ brings the register into normal operating condition. Mode select input S_1 and S_0 defines the various operating modes of the register. $S_1 = 1$ and $S_0 = 0$ makes the register shift-left. $S_1 = 0$ and $S_0 = 1$ makes the register shift-right. $S_1 = 1$ and $S_0 = 1$ loads the parallel data inputs $D_0D_1D_2D_3$ into outputs $D_0D_1D_2D_3$. $S_1 = 0$ and $S_0 = 0$ retains the existing data in hold mode. The transition of S_1 and S_0 inputs from 1 to 0 should be made while clock input $CK = 1$. The clock (CK) of the register is rising-edge-triggered.

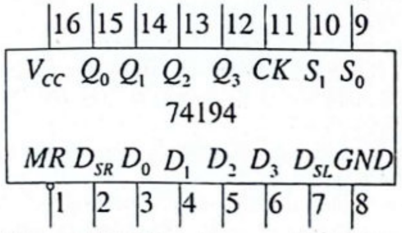


Figure 15.11 Pin assignment of IC 74194

Implementing and testing a 4-bit ring counter using IC type 74194 (4-bit bi-directional universal register)

1. Draw the logic diagram of a 4-bit ring counter using IC type 74194 (4-bit bi-directional universal register).

Lab Work

ICs required

7474 Dual +edge triggered D flip-flops
74194 4-bit bi-directional universal register

Implementing and testing a 4-bit serial-in serial-out right-shift register using D flip-flops

1. Construct the logic diagram of a 4-bit serial-in serial-out right-shift register using IC type 7474 (dual +edge triggered D flip-flops). Connect serial-input (D_R) to a data switch. Also connect Pr and Cr inputs to two data switches. Connect CK input to a push-button pulser. Connect serial-output (Q_R) to an LED indicator.
2. Asynchronously set the output of all flip-flops to 0 by setting $Cr = 0$, $Pr = 1$ and $CK = 0$. Observe the value of the output Q_R.
3. Set $Cr = 1$ for normal operation. Enter data sequence 0110 (the right most bit is the LSB) and observe the output Q_R after application of each clock pulse. Observe both writing and reading processes of the register.

Testing the operation of IC type 74194 (4-bit bi-directional universal register)

1. Connect the master reset input MR', mode selector inputs S_0 and S_1, serial data inputs D_{SL} D_{SL} and D_{SR}, and parallel data inputs D_0, D_1, D_2, and D_3 to nine switches. Connect outputs Q_0, Q_1, Q_2, and Q_3 to four LED indicators. Connect CK input to a push-button pulser.
2. Clear all the outputs Q_0, Q_1, Q_2, and Q_3 by setting master reset input $MR' = 0$. Then set $MR' = 1$ for normal operation.
3. Make the register shift-left by setting the mode selector inputs $S_1 = 1$ and $S_0 = 0$. Enter the serial data 0110 (the left most bit is the LSB) using serial data input D_{SL} and observe the serial data output at Q_3. Also simultaneously observe all the parallel outputs.
4. Make the register shift-right by setting the mode selector inputs $S_1 = 0$ and $S_0 = 1$. Enter serial data 0110 (the right most bit is the LSB) using serial data input D_{SR} and observe the serial data output at Q_0. Also simultaneously observe all the parallel outputs.
5. Make the register parallel-in parallel-out by setting the mode selector inputs $S_1 = 1$ and $S_0 = 1$. Enter the parallel data 0110 (the right most bit id the LSB) using parallel data inputs D_0, D_1, D_2 and D_3. D_0, D_1, D_2 and D_3. Observe the outputs at Q_0, Q_1, Q_2 and Q_3.
6. After parallel-load, make the register both left-shift and right-shift and observe the shifting of the data.
7. Make the register in hold mode by setting the mode selector inputs $S_1 = 0$ and $S_0 = 1$. Observe the outputs after application of some clock pulses. The transition of S_1 and S_0 inputs from 1 to 0 should be made while $CK = 1$.

Implementing and testing a 4-bit ring counter using IC type 74194 (4-bit bi-directional universal register)

1. Construct the logic diagram of a 4-bit ring counter using IC type 74194 (4-bit bi-directional universal register). Connect the master reset input MR', mode selector inputs S_0 and S_1, and parallel data inputs D_0, D_1, D_2, and D_3 to seven switches. Connect outputs Q_0, Q_1, Q_2, and Q_3 to four LED indicators. Connect CK input to a push-button pulser. Set $MR' = 1$ for normal operation.
2. Load 1000 (the right most bit is the LSB) into the register using parallel loading technique.
3. Apply clock pulses to the CK input and observe that the 1 circulates around the outputs.

15.13 Counters

Objectives

1. To implement and test a ÷8 asynchronous up/down counter using J-K flip-flops.
2. To test the operation of MSI IC type 7493 (4-bit binary asynchronous up counter).
3. To design, implement, and test an odd sequence counter using J-K flip-flops.

Theory

Design of asynchronous counters using flip-flops is discussed in Section 11.11. The IC type 7493 (4-bit asynchronous up counter) is introduced in Section 11.15. The design of odd sequence counter is discussed in Section 11.13.

Pre-Lab Work

Implementing and testing a ÷ 8 asynchronous up/down counter using J-K flip-flops

1. Draw the transition diagram of a ÷8 up/down counter.
2. Draw the logic diagram for implementing a ÷8 asynchronous up/down counter using IC type 7476 (dual J-K master-slave flip-flops).

Testing the operation of MSI IC type 7493 (4-bit binary asynchronous up counter)

1. Learn the pin assignment of IC type 7493 as shown in Figure 15.12. The IC has two separate up counters, one ÷2 counter and one ÷8 counter. CK_0 is the clock input and Q_0 is the counter output of the ÷2 counter. CK_1 is the clock input and $Q_3Q_2Q_1$ are the counter outputs of the ÷8 counter. The master reset inputs MR_1 and MR_2 internally connected by a NAND gate to provide the active-LOW clear input to all flip-flops. To make the counter a ÷16 counter, Q_0 is to be externally connected to CK_1.

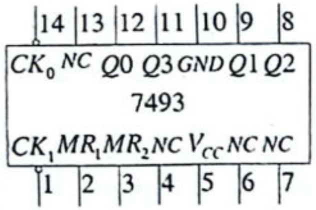

14
13
12
11
10
9
8
CK_0 NC Q0 Q3 GND Q1 Q2
7493
CK_1 MR_1 MR_2 NC V_{CC} NC NC
1
2
3
4
5
6
7

Figure 15.12 Pin assignment of IC type 7493.

Designing, implementing, and testing an odd sequence counter using J-K flip-flops

1. Design an odd sequence counter using J-K flip-flops, which counts the sequence 1-3-5-7 and then 1 to repeat the sequence.
2. Draw the logic diagram for implementing the odd sequence counter using IC type 7476 (dual J-K master-slave flip-flops)

Lab Work

ICs required

7404 Hex NOT gate
7408 Quad 2-input AND gates
7432 Quad 2-input OR gates
7476 Dual J-K master-slave flip-flops
7493 4-bit binary asynchronous up counter

Implementing and testing a ÷8 asynchronous up/down counter using J-K flip-flops

1. Construct the logic diagram for implementing a ÷8 asynchronous up/down counter using IC type 7476 (dual J-K master-slave flip-flops). Connect inputs *M* (mode selector), *Pr*, and *Cr* to three data switches. Connect *CK* input to a push-button pulser. Connect outputs Q_0, Q_1, and Q_2 to the inputs of the 7-segment display decoder/driver on the digital trainer board. Apply a 0 to the MSB of the inputs of the 7-segment display decoder/driver.
2. Asynchronously clear the output of the counter by setting $Cr = 0$, $Pr = 1$, and $CK = 0$. Then set $Cr = 1$ and $Pr = 1$ for normal operation of the counter.
3. Set mode selector for down counting. Apply some clock pulses and observe the output.
4. Set mode selector for up counting. Apply some clock pulses and observe the output.

Testing the operation of MSI IC type 7493 (4-bit binary asynchronous up counter)

1. Connect MR_1 and MR_2 inputs to two data switches. Connect CK_0 clock input to a push-button pulser. Connect CK_1 clock input to Q_0 output. *Q* Connect Q_3, Q_2,

Q_1, and Q_0 outputs to the inputs of the 7-segment display decoder/driver on the digital trainer board.

2. Clear the counter by setting $MR_1 = 1$ and $MR_2 = 1$. Then set any of them to 0 for normal operation of the counter.
3. Apply some clock pulses and observe the output.
4. In a similar manner, test the operation of the two separate ÷2 and ÷8 up counters of the IC type 7493.

Designing, implementing, and testing an odd sequence counter using J-K flip-flops

1. Construct the logic diagram for implementing the odd sequence counter using IC type 7476 (dual J-K master-slave flip-flops). Connect inputs *Pr* and *Cr* to two data switches. Connect *CK* input to a push-button pulser. Connect outputs Q_0, Q_1, and Q_2 to the inputs of the 7-segment display decoder/driver on the digital trainer board. Apply a 0 to the MSB of the inputs of the 7-segment display decoder/driver.
2. Asynchronously set the output of the counter to 111 by setting $Cr = 1$, $Pr = 0$, and $CK = 0$. Then set $Cr = 1$ and $Pr = 1$ for normal operation of the counter.
3. Apply some clock pulses and observe the output.

15.14 Memory Unit

Objectives

1. To test the operation of IC type 7489 (16 × 4 RAM).
2. To simulate a ROM using IC type 7489 and to implement combinational circuits using the simulated ROM.

Theory

IC type 7489 (16 × 4 RAM) is introduced in Figure 12.5. Combinational circuit implementation using PROM is discussed in Section 12.6.

A ROM simulator is obtained from a RAM when operated in the read mode only. The data pattern is first entered into the simulating RAM by placing the unit in the write mode. Then simulation is achieved by placing the unit in the read mode. The simulated ROM can then be used to implement any combinational circuit.

Pre-Lab Work

Testing the operation of IC type 7489 (16 × 4 RAM)

1. Learn the pin assignment of IC type 7498 (16 × 4 RAM) as shown in Figure 15.13. A_3 through A_0 are the four address lines. I_3 through I_0 are the four input data lines and O'_3 through O'_0 are the four output data lines. The memory enable (ME') input must be equal to 0 to enable the memory. If ME' is equal to 1, the memory is disabled and all four outputs are 1. The write enable

(WE) input determines the type of operation as indicated in the function table of Table 15.3. The write operation is performed when $WE = 0$. This is a transfer of the binary number from the data inputs ($I_3I_2I_1I_0$) into the selected word in memory. The read operation is performed when $WE = 1$. This transfers the complement value stored in the selected word into the output data lines ($O'_3O'_2O'_1O'_0$). The outputs are open-collector.

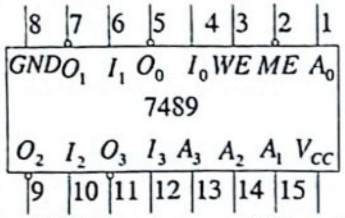


Figure 15.13 Pin assignment of IC type 7498.

Table 15.3 Function table of IC type 7489.

ME' WE	Operation	Data Output
0 0	Write	Complements of data inputs
0 1	Read	Complements of data inputs
1 x	Disable	All 1s

An open-collector gate requires an external resistor for proper operation. However, an open-collector gate can be operated without an external resistor if its output is connected to the input of another gate. Since the outputs of the IC type 7489 produce the complement values, we might as well insert four inverters to change the outputs to their normal value and, at the same time, avoid the need for external resistor.

Simulating a ROM using IC type 7489 (16 × 4 RAM) and implementing combinational circuits using the simulated ROM

1. The combinational circuit to be implemented using the ROM simulator is the 4-bit binary to Gray code converter (Section 2.8). Prepare the truth table of the desired converter.

Lab Work

ICs required

7404 Hex NOT gate
7489 16 × 4 RAM

Testing the operation of IC type 7489 (16 × 4 RAM)

1. Connect the address inputs A_3, A_2, A_1 and A_0 to four data switches. Also connect the address inputs to four LED indicators to see the selected address.
2. Connect the four data inputs I_3, I_2, I_1, and I_0 to four data switches.
3. Connect the four data outputs O'_3, O'_2, O'_1, and O'_0 to four NOT gates. Connect the output of the four NOT gates to four LED indicators.
4. Connect the ME' input to ground and WE input to a data switch.
5. Set the address to 0000. Store an arbitrary 4-bit word into the 0000 location as follows: Set $WE = 1$ (read mode). Set the four data inputs to desired values. To store the word into the selected memory location, set $WE = 0$ (write mode) and then return it to the read mode.
6. Now set the address to 0001 and store another arbitrary word into the memory. In a similar manner, store some words into the memory.
7. Now, by changing the address values, read the stored data and verify them.

Note: Always leave the WE input in the read mode, unless you want to write into memory. Be careful not to change the address or the data inputs when WE is in the write mode.

Simulating a ROM using IC type 7489 (16 × 4 RAM) and implementing combinational circuits using the simulated ROM

1. Store the truth table of the 4-bit binary number to Gray code converter into the IC type 7489 RAM by setting the address inputs to the binary value and the data inputs to the corresponding Gray code value.
2. After all 16 entries of the truth table are written into memory, set the ROM simulator by connecting the WE to 1 permanently.
3. Check the code converter by applying the inputs to the address lines and verifying the correct outputs in the data output lines.

15.15 Verilog Simulation of a 4-bit Parallel Adder

Objectives

1. To model a 4-bit parallel adder circuit using Verilog hardware description language.
2. To simulate the Verilog description for verifying the correctness of the model.

Theory

Binary parallel adder is introduced in Section 7.4. Introduction to Verilog description and simulation is discussed in Section 14.1. Details of behavioral modeling of combinational circuits are discussed in Sections 14.3 and 14.4.

Pre-Lab Work

1. Write a module for structural description of a half-adder.
2. Write a module for a full-adder by instantiating half-adder module.
3. Write a module for a 4-bit parallel adder by instantiating full-adder module.
4. Write a test generator for testing the 4-bit parallel adder.
5. Write a top-level testbench module for the system.

Lab Work

Equipment required

1. A PC with Verilog simulator installed.

Procedure

1. Using a suitable editor, write the modules designed in the pre-lab work and save the files in the appropriate folder (if necessary, consult your lab technicians for more details of simulator environment).
2. Build a project for the system and simulate it. Verify the output of the simulation.
3. If simulation results are not correct, make appropriate corrections in your module descriptions and re-simulate it until the results are correct.

15.16 Verilog Simulation of an Up/Down BCD Counter

Objectives

1. To model an up/down BCD counter using Verilog hardware description language.
2. To simulate the Verilog description for verifying the correctness of the model.

Theory

Design of counters is introduced in Sections 11.11 and 11.12. Introduction to Verilog description and simulation is discussed in Section 14.1. Details of behavioral modeling of sequential circuits are discussed in Sections 14.5.

Pre-Lab Work

1. Write a module for behavioral description of a clock generator.
2. Write a module for behavioral description of an up/down BCD counter.
3. Write a top-level module by instantiating the clock generator module and the BCD counter module for simulation.

Lab Work

Equipment required

1. A PC with Verilog simulator installed.

Procedure

1. Using a suitable editor, write the modules designed in the pre-lab work and save the files in the appropriate folder (if necessary, consult your lab technicians for more details of simulator environment).
2. Build a project for the system and simulate it. Verify the output of the simulation.
3. If simulation results are not correct, make appropriate corrections in your module descriptions and re-simulate it until the results are correct.

Prof. Dr. Md. Mozammel Huq Azad Khan is currently the Dean of Faculty of Sciences and Engineering at East West University, Dhaka. He has obtained B. Sc. Engg. degree in Electrical and Electronic Engineering, M. Sc. Engg. degree in Computer Engineering, and Ph. D degree in Computer Science and Engineering. He is one of the pioneering professors of Computer Science and Engineering in Bangladesh. He started his teaching career in Electrical and Electronic Engineering Department at Bangladesh Institute of Technology (BIT), Rajshahi in 1986. In 1991 he shifted to Khulna University and, as the founder Head of the Department, established Computer Science and Engineering Department there, which was the second of its kind in the country after that of Bangladesh University of Engineering and Technology (BUET), Dhaka. He was also the founder Head of Electronics and Communications Engineering Department and also served as Dean of Science, Engineering and Technology School at Khulna University. In 1999 he joined East West University. He served as the Chairperson of Computer Science and Engineering Department and also was the founder Chairperson of Electrical and Electronic Engineering Department at East West University. He is a renowned researcher in the broad area of digital logic synthesis and optimization in home and abroad.